VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Hadoop a Spark pro zpracování dat na HPC infrastruktuře

# Hadoop and Spark for data processing on the HPC infrastructure

2018 Bc. Jiří Cága

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Jiří Cága**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Využití Spark pro zpracování dat na HPC infrastruktuře
Spark for Data Processing on the HPC Infrastructure

Jazyk vypracování: čeština

Zásady pro vypracování:

V rámci diplomové práce se student seznámí s technologiemi Apache Spark. Pomocí těchto technologií student následně provede implementaci vybraného machine learning algoritmu a provede jejich testování nad reálnými daty v prostředí HPC clusteru. Další části diplomové práce bude prostudování a navržení využití Spark v aplikačním rámci HPC as a Service, který je vyvíjen v rámci centra IT4Innovations. Výsledkem bude rozšíření rámce HPC as a Service o knihovnu umožňující spouštění a monitorování HPC jobů využívajících Spark.

Jednotlivé body zadání jsou:
1. Prostudování technologií Apache Hadoop a Apache Spark.
2. Prostudování a otestování možností Spark na výpočetním clusteru.
3. Rozšíření aplikačního rámci HPC as a Service o volání Spark.
4. Implementace vybraného machine learning algoritmu a jeho otestování v rámci vyvinutého řešení.
5. Provedení experimentů a jejich vyhodnocení.

Seznam doporučené odborné literatury:

[1] Donald Miner, Adam Shook: MapReduce Design Patterns, O'Reilly Media, December 2012. ISBN 978-1-4493-2717-0.
[2] Hadoop Tutorial [online 2015-11-01], http://www.tutorialspoint.com/hadoop/index.htm
[3] History of Hadoop. GIGAOM. [online 2015-11-01], https://gigaom.com/2013/03/04/the-history-of-hadoop-from-4-nodes-to-the- future-of-data/

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Martinovič, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 30.04.2018


| | |
|---|---|
| doc. Ing. Jan Platoš, Ph.D. | prof. Ing. Pavel Brandštetter, CSc. |
| *vedoucí katedry* | *děkan fakulty* |

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 13. dubna 2018

.....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 13. dubna 2018 .....

Rád bych na tomto místě poděkoval vedoucímu diplomové práce Ing. Jan Martinovič, Ph.D. za rady v oblasti distribuovaných výpočtů a panu Ing. Václavu Svatoňovi za ochotu vysvětlení principu služby HPC as a Service.

**Abstrakt**

Diplomová práce popisuje technologie Apache Hadoop a Spark. V první částí seznamuje jak s popisem technologií tak s implementací vybraných algoritmů za pomoci těchto technologií. Druhá část je věnována návrhu grafického klienta pro spouštění implementovaných algoritmů nad službou HPC as a Service. Hlavním cílem bylo porovnání různých implementací algoritmů s využitím Apache Hadoopu a Sparku nad rozsáhlými datovými sadami na infrastruktuře HPC v technologickém centru IT4Innovations.

**Klíčová slova**: HPC, Hadoop, Spark, Machine learning algoritmy, Měření paralelismu

**Abstract**

Diploma thesis describes technologies an Apache Hadoop and a Spark. In first part it explains technologies and implementation selected algorithms. The second part is devode design graphic client for launching implemented algorithms on HPC as a Service. The main goal was compare different implementation algorithms with use Hadoop and Spark onto range of dataset on HPC infrastructure in technology center IT4Innovations.

**Key Words**: HPC, Hadoop, Spark, Machine learning, Paralelism measure

# Obsah

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| HPC | – | High Performance Computing |
| SSD | – | Solid State Disc |
| FLOPS | – | Floating Point Operations Per Second |
| PBS | – | Portable Bash System |
| SSH | – | Secure Shell |
| RDD | – | Resilient Distributed Dataset |
| SQL | – | Structured Query Language |
| JDBC | – | Java Database Connectivity |
| ODBC | – | Open Database Connectivity |
| SOAP | – | Simple Object Access Protocol |
| JVM | – | Java Virtual Machine |
| WSDL | – | Web Services Description Language |
| HTML | – | Hyper Text Markup Language |
| XML | – | Extensible Markup Language |
| CSS | – | Cascading Style Sheets |
| MVC | – | Model View Control |
| SAX | – | Simple API for XML |

# Seznam obrázků

## Seznam tabulek

# Seznam výpisů zdrojového kódu

# 1 Úvod

Dnešní svět je plný proudících dat kolem nás. Každým rokem se objem dat zvětšuje, a dle studie IDC digital Universe[1] objem dat během následujících dvou let vzroste minimálně dvakrát. Data se sbírají při nákupů zboží v internetových obchodech, při volání z mobilního telefonu, placením platební kartou a také při využívání elektronických komunikačních prostředků jako je email či sociální sítě. Rovněž si lze všimnout stále oblíbenějšímu ukládání dat do cloudových služeb, kde uživatelé nahrávají celé své fotoarchivy a videa. A to jsou pouze data vytvořená člověkem. Spoustu dalších dat vyprodukují aplikace jako záznam uživatelských preferencí při návštěvě internetové stránky, ukládání názvu programu zapisujícího do databáze či archivace logů o chybách serveru. Data se následně snažíme shromažďovat a analyzovat pro získání cenných informací. Informace nám můžou poskytnout konkurenční výhodu, například internetový obchod analyzující chování zákazníka mu může doporučit další vhodný produkt ke koupi na základě jeho historie objednávek. Podobný scénář využívají i mobilní operátoři analyzující, jak dlouho voláme, kolik napíšeme textových zpráv a na základě toho nám nabízejí tarif. Další příklad může být v lékařství kde sbíráním příznaků nemocí můžeme přesněji určit diagnozu pacienta[1].

Problém dnešní doby je velké množství nově vznikajících dat, které už nelze analyzovat běžnými postupy, například v tabulkovém procesoru nebo relační databázi. Proto vzniklo nové odvětví nazývané Big Data. Abychom mohli data označit jako Big Data musí splňovat následující tři podmínky: Volume (příliš velké množství dat), Velocity (velká rychlost vniku nových dat), Variety (různé formáty dat, strukturované formáty spolu s nestrukturovanými)[2]. Pro zpracování těchto dat potřebujeme velký výkon, který lze získat například spojením několika výpočetních uzlů pro distribuované výpočty. K tomu můžeme využít technologie od firmy Apache, ať už Hadoop[3] nebo Spark[4], umožňující distribuované zpracování velkých datových sad přes výpočetní uzly počítačů pomocí jednoduchého programového modelu. Obě technologie jsou navrženy pro snadnou škálovatelnost z jednotek výpočetních uzlů na několik tisíc výpočetních uzlů, navíc poskytují mechanizmus k detekci a zvládnutí výpadku jednoho z výpočetních uzlů díky replikaci dat[3]. A právě těmito technologiemi se budeme dále zabývat v diplomové práci.

[1] Informace převzata z https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

## 1.1 Struktura práce

V první části práce bude čtenář seznámen s architekturou superpočítačů Anselm a Salomon v prostředí IT4Innovations[2]. Poté následuje popis technologií Hadoop a Spark od firmy Apache s jejich nasazením na vybraný superpočítač. Dále se v práci popisuje implementace dvou algoritmů nad modelem MapReduce[3], který je vhodný pro distribuovaný výpočet a vnitřně používaný u technologie Hadoop a Spark. Pro textovou analýzu byl zvolen algoritmus Invertovaného indexování a pro shlukovou analýzu machine learning algoritmus K-means. Druhá část práce je věnovaná návrhu grafického klienta spouštějícího algoritmy implementované v této práci přes službu HPC as a Service. V poslední části práce je uvedeno porovnání různých implementací algoritmů ve výše zmíněných technologiích nad rozsáhlými datovými sadami s využitím infrastruktury HPC v technologickém centru IT4Innovations.

[2] Bližší informace o superpočítačovém centru naleznete na: https://www.it4i.cz
[3] Popis fungování modelu MapReduce naleznete na: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

# 2 Architektura superpočítače

Superpočítač je výpočetní systém složený z hardwaru, systémového a aplikačního softwaru, který poskytuje nejlepší dosažitelný výkon pro náročné výpočetní problémy. Výpočetní výkon se mezi superpočítači porovnává pomocí jednotky zvané FLOPS v překladu znamenající počet operací v plovoucí řadové čárce za sekundu [5].

V technologickém centru IT4Innovations jsou dva superpočítače pojmenované Anselm a Salomon, přičemž nejvýkonnější Salomon dosahuje max. teoretického výkonu 2 PFLOPS[15]. Při porovnání s dnes běžným počítačem s výkonem okolo stovek GFLOPS[4] je Salomon přibližně 10000x rychlejší. Anselm ani Salomon nejsou ve skutečnosti jeden počítač, ale jsou složeny z několika výpočetních uzlů. Každý výpočetní uzel obsahuje speciální serverový procesor Intel Xeon a velkou operační paměť s kapacitou několik desítek GB. Aby výpočetní uzly spolu mohly rychle komunikovat a nedocházelo ke zpoždění jsou propojeny sítí typu InfiniBand[5] vyznačující se vysokou rychlostí a malou latencí.[15] Jedná se o levnější řešení než výroba jediného počítače o stejném výkonu. Při programování algoritmů pro HPC je nutné použít jiné postupy než při programování algoritmů pro osobní počítače. Důležité je zajistit možné rozdělení výpočtu algoritmu na několik částí, které se přerozdělí mezi jednotlivé výpočetní uzly.

## 2.1 Anselm

Superpočítač Anselm je tvořen 209 výpočetními uzly s celkem 3344 procesorovými jádry. Celková operační paměť dosahuje kapacity 15 TB. Superpočítač je schopný teoretického výkonu 94 TFLOPS. Výpočetní uzly jsou vybaveny procesorem s 16 jádry postavený na architektuře x86-64 , operační pamětí o velikosti 64GB a 500GB pevným diskem. Celkem 23 výpočetních uzlů obsahuje grafický akcelerátor NVIDIA Kepler. Na výpočetních uzlech běží operační systém Bullx Linux obsahující podporu pro HPC infrastrukturu. Bullx se velmi podobá Linuxu z rodiny RedHat[6]. Jednotlivý uživatelé mají vyčleněnou kapacitu pro svá data (složka home, 320GB) a společný pracovní prostor všech uživatelů (SCRATCH, 146TB). Ke spouštění úloh se používá plánovací manager PBS zmíněný dále v textu[15].

## 2.2 Salomon

Superpočítač je tvořen 1008 výpočetními uzly s celkem 24192 procesorovými jádry. Celková operační paměť dosahuje kapacity 129 TB. Superpočítač je schopný teoretického výkonu 2 PFLOPS. Výpočetní uzly jsou vybavené procesorem s 24 jádry postavený na architektuře x86-64, operační pamětí o kapacitě 128GB. Celkem 432 výpočetních uzlů je vybavena akcelerátorem Xeon Phi MIC.

[4] Popis procesoru Intel core i7: https://techgage.com/article/intels-skylake-core-i7-6700k-a-performance-look/
[5] Popis sítě InfiniBand naleznete na: http://www.hpcadvisorycouncil.com/pdf/Intro_to_InfiniBand.pdf
[6] Dokumentace k RedHatu: https://www.redhat.com/en/technologies/linux-platforms

Obrázek 1: Pohled na hardware Anselmu[15]

Na výpočetních uzlech běží operační systém CentOS[7] obsahující podporu pro HPC infrastrukturu. CentOS se velmi podobá Linuxu z rodiny RedHat. Všechny výpočetní uzly sdílejí 0,5 PB disk sloužící pro ukládáni uživatelských dat (adresář HOME) a společný pracovní prostor všech uživatelů (SCRATCH, 1,6PB). Úlohy pro superpočítač se nejdříve jako u Anselmu předloží PBS managerovi zajišťující spouštění úlohy na dostupných zdrojích[15].

## 2.3 Plánování spouštění úloh

K superpočítači přistupuje najednou více uživatelů a každý z nich chce provádět výpočty. Výpočty se nemůžou ihned spouštět, došlo by k rychlému využití celého výkonu počítače a nikdo dál by s ním nemohl pracovat. Proto se na superpočítačích Anselm a Saloman používá HPC job scheluder starající se o řízení a monitorování celkového zatížení výpočetních uzlů. Oba superpočítače používají PBS scheluder, což je komerční implementace HPC job scheluderu[16].

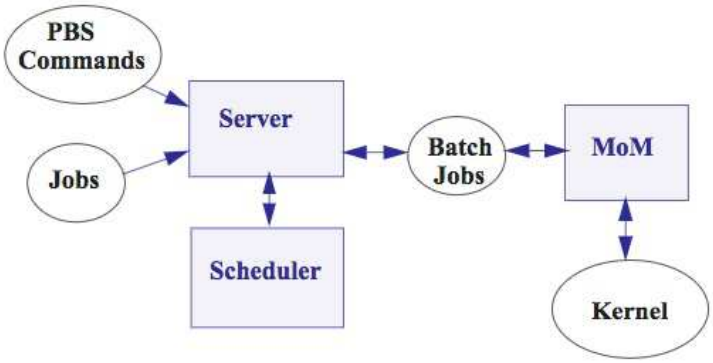
Cílem scheluderu je plnit hlavně tyto tři funkce:

- **Řazení do fronty**: hlavním cílem scheluderu je sběr všech výpočetních úkolů určených k běhu na superpočítači. Uživatelé předloží nové úlohy a scheluder udržuje všechny úlohy ve frontě až do uvolnění zdrojů pro spouštění další úlohy.
- **Plánovaní**: pro efektivní zpracování výpočetních úloh musí scheluder umět řídit kde a kdy poběží úloha dle předurčených pravidel. Například jeden uživatel může najednou využít jenom 20 procent výpočetního výkonu.
- **Monitorování**: získávání informací o dostupnosti a vytížení zdrojů, dovoluje scheluderu správně naplánovat spouštění další úlohy z fronty.

[7] Dokumentace k systému CentOS: https://www.centos.org

### Architektura scheluderu PBS

Jak už bylo zmíněno oba superpočítače v technologickém centru IT4Innovations používají komerční implementaci scheluderu PBS, proto se následující odstavce budou zabývat podrobnějším fungováním tohoto scheluderu.




Obrázek 2: PBS architektura[16]

- **Commands**: skupina příkazů poskytovaná PBS scheluderem pro předložení,monitorování, úpravu nebo smazání úlohy. Příkazy můžeme dělit do dvou skupin, do první skupiny patří příkazy možné vykonávat PBS uživatelem a do druhé příkazy možné volat pouze administrátory.
- **Job**: skript obsahující příkazy, které chceme přes PBS scheluder spustit. Skript lze napsat v Bashy nebo jazyku Python a následně předložit komponentě PBS server pomocí příkazu *qsub*[8]
- **Server**: proces běžící na pozadí provádějící správu PBS jobů. Po předložení úlohy příkazem qsub je úloha zařazena do fronty. Ve frontě čeká než rozhodne komponenta Scheluder o jejím vykonání.
- **Scheluder**: proces běžící na pozadí provádějící monitorování volných zdrojů a nastavení pravidel dané úloze. Například kdy bude úloha vybrána z fronty a předložena pro vykonání komponentě MoM.
- **MoM**:proces běžící na pozadí, též označovaný Job Executor, zodpovědný za provedení úlohy, jakmile je přeposlána z komponenty Server[16].

[8]Popis příkazu qsub: http://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm

### Předložení úlohy PBS scheluderu

Pokud chceme na superpočítači Anselm nebo Salomon spustit výpočet, musíme nejdříve svoji úlohu zaregistrovat u PBS scheluderu. Registraci lze provést příkazem *qsub* pomocí PBS skriptu nebo jednořádkovým příkazem z konzole.

#### PBS skript

PBS skript lze napsat v Bashy[9] nebo programovacím jazyku Pythonu. Ve výpisu 1 je ukázka skriptu v Bashy. První řádka skriptu identifikuje shell používaný uživatelem, v tomto případě se jedná o Bash. Na druhé řádce je určen počet uzlů a procesorů v jednotlivých uzlech. Třetí řádka udává kolik času by měla úloha vyžadovat. Čtvrtý řádek říká kde jsou uložena data pro předložený algoritmus. Pátý řádek obsahuje cestu k programu. Na posledním řádku jsou uvedené parametry programu.

```
#!/bin/bash
#PBS -l nodes=1:ppn=2
#PBS -l walltime=00:00:59
cd /home/rcf-proj3/pv/test/
./myJob.sh
```

Výpis 1: Ukázka PBS skript

#### Jednořádkový příkaz

Úlohu lze jednoduše PBS scheluderu předložit i pomocí jednoho příkazu v terminálu(viz. Výpis 2). V příkazu nejdříve specifikujeme frontu, kde úloha poběží. Poté specifikujeme počet výpočetních uzlů a počet procesorů na každém z uzlu. Nakonec se přidá cesta k úloze, kterou chceme spustit. Příkaz obsahuje řadu dalších parametrů, jako například maximální dobu běhu úlohy, které můžete nalézt v dokumentaci[10].

```
qsub -q qexp -l select=4:ncpus=24 /home/rcf-proj3/pv/test/myjob.sh
```

Výpis 2: Ukázka QSUB příkaz

[9] Článek seznamující s Bash shellem: https://www.root.cz/clanky/programovani-v-bash-shellu/
[10] Dokumentace qsub: http://gridscheduler.sourceforge.net/htmlman/htmlman1/qsub.html

# 3 Technologie Hadoop

Kapitola seznamující s Hadoopem je rozdělena do řady podkapitol, nejdříve je vysvětleno co je Hadoop s krátkým úvodem do jeho historie. Poté následuje popis přístupu k datům u tradičních informačních systémů a jak k datům přistupuje Hadoop. V další kapitole je rozebrána architektura Hadoopu s podrobným vysvětlením fungování algoritmu MapReduce. Teoretická část zakončuje rozbor fungování distribuovaného diskového formátu HDFS. Poslední kapitola vysvětluje nasazení Hadoopu na superpočítač Salomon.

### Definice

Hadoop je open source framework od firmy Apache napsaný v programovacím jazyku Java. Framework je určený k provádění operací nad velkým množstvím dat a lze nasadit na skupinu výpočetních uzlů tvořících systém s distribuovaným výpočetním výkonem a úložnou kapacitou. Navíc Hadoop umožňuje jednoduché rozšíření z jednoho výpočetního uzlu na několik tisíc výpočetních uzlů, přičemž každý výpočetní uzel poskytuje svůj lokální výkon a diskový prostor[3]

### Historie

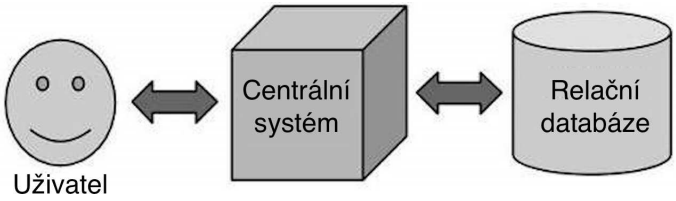
- **2002** Dough Cutting a student Mike Cafarella vytvořili open source vyhledávací nástroj Nutch.
- **2003** Společnost Google uvolnila informace ohledně Google File System.
- **2004** Společnost Google uvolnila informace ohledně MapReduce algoritmu.
- **2006** Dough Cutting upravil svůj nástroj Nutch pro vyhledávání, přidal podporu MapReduce algoritmu a souborový systém podobný Google File Systému, čímž vznikl framework Hadoop.
- **2015** Téměř všechny populární weby (Facebook, eBay, Twitter) používají Hadoop k analyzování velkého množství dat generovaných uživateli[6].

## 3.1 Přístup k datům

V následující kapitole se porovnává přístup k datům u tradičního podnikového systému s přístupem k datům pomocí algoritmu MapReduce využívaného u frameworku Hadoop.

### Tradiční podnikový přístup

Tradiční informační systémy k ukládání dat používají většinou relační databázi (například MS-SQL[11] nebo Oracle DB[12]). Uživatel komunikuje s aplikací a ta následně odesílá požadavky na databázový server. Ukázku komunikace můžete vidět na Obrázku 3.



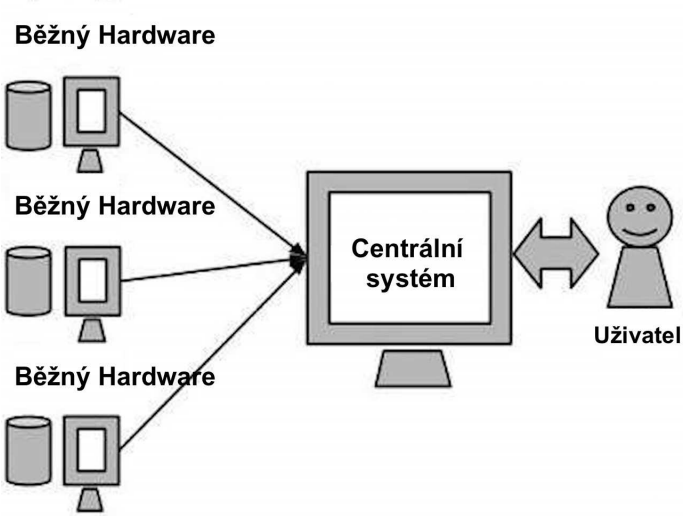

Obrázek 3: Přístup k datům - tradiční systém [3]

Tento přístup funguje dobře tam, kde máme menší objem dat, které mohou být uloženy standardními databázovými servery nebo až do limitu procesoru zpracovávající data. Ale pokud jde o řešení velkého množství dat, je to opravdu zdlouhavý úkol zpracovávat data prostřednictvím databázového serveru[6].

### Google řešení

Řešení od firmy Google se snaží zrychlit zpracování dat díky algoritmu MapReduce, který rozdělí úlohu na menší izolované části jenž se můžou zpracovávat odděleně na více výpočetních uzlech v síti. Zmíněný přístup sbírá v průběhu práce mezivýsledky algoritmu MapReduce a nakonec je integruje do celkového výsledku. Podrobně je algoritmus popsán v následující kapitole. Komunikaci uživatele se systémem můžete vidět na Obrázku 4. Největší výhodou tohoto přístupu je možnost jednoduše rozšiřovat výpočetní výkon přidáním dalších počítačů do clusteru. Počítače ani nemusí být stejného typu, například jeden z připojených může být výkonný server s více procesory a další osobní počítače s jedním procesorem. Nevýhodou může být složité nakonfigurování počítačů na začátku[6].

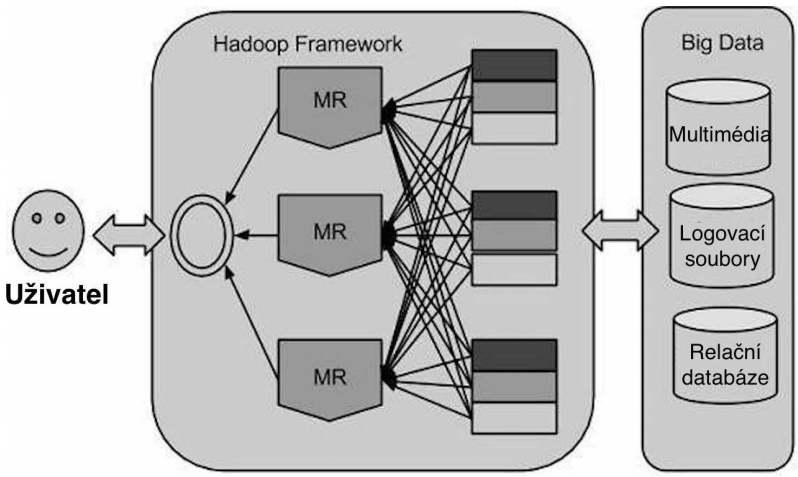
[11] Dokumentace k databázi MS-SQL: https://www.microsoft.com/cs-cz/sql-server/sql-server-2016
[12] Dokumentace k databázi Oracle DB: https://www.oracle.com/database/index.html




Obrázek 4: Přístup k datům - Google řešení [3]

### Hadoop řešení

Doug Cutting, Mike Cafarella a jejich tým začal v roce 2005 vyvíjet framework zvaný Hadoop, který staví nad zmíněným Google řešením. Uvnitř Hadoopu běží algoritmus MapReduce umožňující škálování úlohy na několik počítačů. Hadoop umožňuje přístup k různým datovým zdrojům ať už se jedná o SQL databáze, log soubory či nestrukturovaná data. Ukázku práce uživatele s Hadoopem zobrazuje obrázek číslo 5[6].




Obrázek 5: Přístup k datům - Hadoop ekosystém [3]

## 3.2 Architektura

Framework Hadoop je rozdělen do čtyř modulů:

- **Commmon** – Základní modul celého frameworku obsahující knihovny a nástroje pro spouštění a řízení Hadoopu[3].
- **YARN** - Modul zodpovědný za plánování práce a monitorování volných zdrojů na výpočetních uzlech[7].
- **HDFS** - Distribuovaný souborový systém navržený na běh na klasickém hardwaru s vysokou odolností vůči selhání.[8].
- **MapReduce** - Algoritmus pro provádění distribuovaného výpočtu na několika výpočetních uzlech[9].

## 3.3 MapReduce

MapReduce je algoritmus pro zpracování velkého množství dat distribuovaně na několika výpočetních uzlech. Algoritmus umožňuje snadné horizontální škálování z jednoho výpočetního uzlu na několik set. Proces zpracování algoritmu se skládá ze dvou částí, které mají definované pořadí a nesmí být prohozeny.

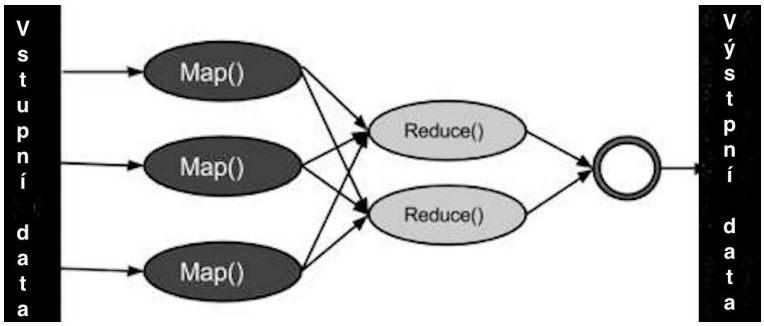
- **První část Map**: načítá množinu dat a převede elementy do entic (klíč, množina hodnot). Nejčastěji se načítání provádí po řádcích z textového souboru uloženého v distribuovaném diskovém úložišti formátovaného v HDFS.
- **Druhá část Reduce**: vezme předem připravené data z části Map a provede transformaci entice na menší entici a uloží výsledek. Výsledek bývá nejčastěji ukládán rovněž do distribuovaného diskového úložiště pro další zpracování.

Během provádění výpočtu Hadoop odesílá Map a Reduce úlohy na jednotlivé výpočetní uzly v clusteru. Výpočet se provádí na lokálních discích uzlu, což snižuje zatížení sítě. Po dokončení zadané úlohy odesílá uzel výsledek zpět na Hadoop server[9].

### Fungování MapReduce v Hadoopu

Hadoop rozdělí vstupní data na několik oddělených částí a rozmístí je mezi výpočetní uzly, přičemž jeden výpočetní uzel je označen jako řídící (master). Řídící uzel provádí přerozdělování částí dat a spouštění algoritmu MapReduce na ostatních výpočetních uzlech. Řídící uzel očekává pravidelné odezvy podřízených výpočetních uzlů s informací o průběhu výpočtu po zadání práce. V případě, že se podřízený uzel dlouho neozývá, považuje ho hlavní uzel jako neaktivní a přeřadí zpracování úlohy na jiný dostupný výpočetní uzel. Dále se snaží hlavní uzel ponechat obě části Map a Reduce na stejném podřízeném výpočetním uzlu, aby nedocházelo k vytížení

sítě. Hlavní uzel je rovněž zodpovědný za bezpečnost atomických operací. Mezi atomické operace patří například zápis do souboru, a nemělo by se stát, aby dva výpočetní uzly zapisovaly do stejného souboru ve stejný čas[3].




Obrázek 6: Diagram algoritmu MapReduce[9]

## 3.4 Příklady použití MapReduce

V následující kapitole jsou popsány dva jednoduché příklady využití algoritmu MapReduce v praxi.

### Čítač slov

Algoritmus MapReduce začne zpracování funkcí Map, která rozdělí dokument na jednotlivá slova. Výstupem funkce Map je entice, kde slovo představuje klič a jako hodnota se ukládá číslo jedna. Jednička symbolizuje jeden výskyt daného slova. Poté algoritmus seskupí všechny hodnoty se stejným klíčem do jedné kolekce a nad touto kolekcí zavolá funkci Reduce. Funkce Reduce sečte celkový počet záznamů v kolekci pro určité slovo a tento počet uloží jako celkový počet slov v dokumentu.

```
// name: document name, document: document contents
function map(String name, String document):
 for each word w in document:
   emit (w, 1)

// word: a word, partialCounts: a list of aggregated partial counts
function reduce(String word, Iterator partialCounts):
 sum = 0
 for each pc in partialCounts:
   sum += pc
 emit (word, sum)
```

Výpis 3: Zdrojový kód čítače slov dle MapReduce

### Operace nad velkou tabulkou

Mějme tabulku v databázi s 1,1 bilionem záznamů lidí a chceme spočítat průměrný počet kontaktů lidí dle věku. Jedná se o časově velmi náročnou operaci,ale můžeme ji provést pomocí paralelního zpracování algoritmem MapReduce.

```
SELECT age, AVG(contacts)
  FROM social.person
GROUP BY age
ORDER BY age
```

Výpis 4: SQL dotaz nad velkou tabulkou.

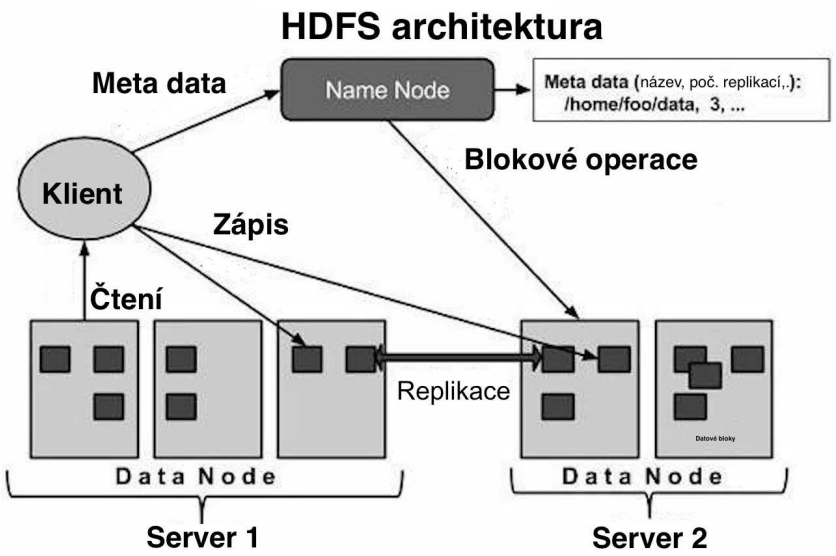
Před spouštěním algoritmu MapReduce se nejdříve data z tabulky rozdělí na menší části mezi jednotlivé výpočetní uzlu. Algoritmus MapReduce začíná funkcí Map, která si do výstupní entice ukládá rok jako klíč a hodnotu představující dvojici (počet kontaktů dané osoby a číslo jedna značící, že se jedná o jednu osobu). Po provedení funkce Map, algoritmus provede spojení záznamů se stejným klíčem nad všemi enticemi. Vznikne nová entice, kde klíč zůstane stejný a jako hodnota se uloží seznam entic patřící k danému klíči. Funkce Reduce následně prochází všechny uložené entice pro daný rok a provede součet všech prvních prvků dvojice (celkové množství přátel lidí) a součet všech druhých prvků dvojice (celkový počet osob). Výstup tvoří klíč představující rok a průměrný počet kontaktů osob v daném roce (celkové množství přátel / celkový počet osob).

## 3.5 HDFS (Hadoop File system)

Distribuovaný souborový systém vhodný pro běh na klasických pevných discích. Mezi jeho hlavní vlastnosti patří vysoká chybová tolerance díky nastavitelné replikaci dat. Souborový systém umožňuje jednoduchý přístup k datům příkazy, které jsou velmi podobné příkazům pro správu souborů v operačním systému Linux. Celkový běh zajišťují uzly NameNode a DataNode[13] zobrazeny v obrázku číslo 7[8].

- **Blok**: data ukládaná do HDFS jsou nejprve rozdělena na jeden čí několik bloků, blok je nejmenší datová jednotka v souborovém systému HDFS. Základní velikost bloku je 64 MB, ale v případě potřeby si může uživatel nastavit i větší velikost.
- **DataNode**: proces běžící na pozadí, jenž zajišťuje základní souborovou práci jako je čtení a zápis bloku do souborového sytému HDFS. Tento proces běží na několika počítačích, které poskytují svoji diskovou kapacitu a celkově vytváří jeden distribuovaný disk. Jednotlivé DataNode uzly pracují odděleně a nekomunikují meze sebou. Dále je DataNode zodpovědný za replikaci jednotlivých bloků dle informace od uzlu NameNode.

[13] Rozdíl mezi uzly NameNode a DataNode: http://hadoopinrealworld.com/namenode-and-datanode/




Obrázek 7: Schéma HDFS architektury[3]

- **NameNode**: nejdůležitější uzel sloužící jako vstupní bod do souborového systému HDFS. Udržuje informace jak přistoupit k dalším datům. Jeho hlavním úkolem je mapování rozložení jednotlivých DataNode uzlů a pamatovat si kde se jaký datový blok nachází. Jedná se o velmi důležitý uzel, proto Hadoop vytváří jeho kopii pro případ výpadku hlavního NameNode uzlu[8].

## 3.6 Nasazení Hadoop

V této sekci bude popsané nasazení technologie Hadoop na superpočítač Anselm. Původně byl záměr nasadit Hadoop na superpočítač Salomon, kvůli jeho vyššímu výkonu oproti superpočítači Anselm. Nakonec se musel zvolit superpočítač Anselm, protože framework Hadoop vyžaduje lokální diskové úložiště pro každý výpočetní uzel. Tento předpoklad splňuje pouze superpočítač Anselm, zde opravdu každý výpočetní uzel obsahuje svůj lokální disk s kapacitou 330GB. Superpočítač Salomon má sdílené úložiště pro všechny výpočetní uzly, a proto je pro Hadoop nevhodný. Níže zmíněný postup vyžaduje, aby čtenář znal operační systém Linux minimálně na úrovni práce s Bash terminálem[14], protože celý postup nasazení probíhá přes terminálovou konzoli přistupující vzdáleně na superpočítač.

### Přihlášení k superpočítače

Pro přihlášení k superpočítači se používá SSH protokol, pokud používáte Linux zadejte do příkazové řádky níže zmíněný příkaz z Výpisu 5, následně budete vyzvání k zadání uživatelského jména a hesla. Po zadání přihlašovacích údajů se vzdáleně přihlásíte pod svým účtem k superpočítači a můžete zadávat příkazy. Pokud používáte operační systém Windows na svém klientské počítači můžete pro přihlášení použít program WinSCP[15] nebo Putty[16]. K přihlášení je třeba vygenerovaný privátní klíč. Podrobný návod jak získat přístup a vygenerovat klíč můžete nalézt v dokumentaci IT4Innovations[17].

```
ssh -i id_rsa yourUsername@anselm.it4i.cz
```

Výpis 5: Přihlášení do superpočítače

Po úspěšném přihlášení byste měli vidět zprávu v terminálu podobné této:


Obrázek 8: Zpráva po přihlášení Anselm

[14] Dokumentace k Bash: https://www.gnu.org/software/bash/
[15] Dokumentace k WinSCP: https://winscp.net/eng/docs/start
[16] Dokumentace k Putty: https://www.chiark.greenend.org.uk/ sgtatham/putty/docs.html
[17] Generování klíče: https://docs.it4i.cz/anselm/shell-and-data-access/

### 3.6.1 Příprava prostředí

Superpočítač Anselm nemá nainstalovanou žádnou verzi frameworku Hadoop ve výchozím stavu. Nejprve je nutné stáhnout Hadoop a všechny potřebné balíky k jeho práci. Pro nasazení byla zvolena ověřená verze Hadoopu ve verzi 1.2.1. Důvodem byla existence knihovny myHadoop[18] pro tuto verzi Hadoopu, která umožňuje úpravu konfigurace Hadoopu pro svět HPC. Při konfiguraci se osvědčila technika umístit Hadoop a všechny potřebné knihovny do složky hadoop-stack a uvnitř pak provést následné nastavení.

#### Stažení potřebných knihoven

Následující sekce popisuje kroky ke stažení Hadoopu a potřebných balíků pro běh na superpočítači Anselm do složky hadoop-stack. Zmíněné kroky je nutné provést v definovaném pořadí.

1. Vytvoření složky Hadoop-stack

```
mkdir ~/hadoop-stack
```

Výpis 6: Vytvoření složky Hadoop-stack

2. Stažení jdk 7

```
cd ~/hadoop-stack
wget --no-check-certificate --no-cookies --header "Cookie:
    oraclelicense=accept-securebackup-cookie" http://download.oracle.com
    /otn-pub/java/jdk/7u79-b15/jdk-7u79-linux-x64.tar.gz
tar -zxvf jdk-7u79-linux-x64.tar.gz
```

Výpis 7: Stažení Javy

3. Stažení Hadoop 1.2.1

```
cd ~/hadoop-stack
wget https://archive.apache.org/dist/hadoop/core/hadoop-1.2.1/hadoop
    -1.2.1-bin.tar.gz
tar -zxvf hadoop-1.2.1-bin.tar.gz
```

Výpis 8: Stažení Hadoop 1.2.1

4. Stažení myHadoop

```
cd ~/hadoop-stack
git clone https://github.com/glennklockwood/myhadoop.git
```

Výpis 9: Klon repozitáře myHadoop

[18] Článek o nasazení myHadoop v HPC: http://www.sdsc.edu/ allans/MyHadoop.pdf

### Úprava Hadoopu

Pro správnou funkčnost Hadoopu na superpočítači je nutné upravit konfiguraci Hadoopu podle myHadoop šablony. Celý proces můžete provést velmi jednoduše pomocí příkazů z Výpisu 10. Následně bude popsané jaké soubory šablona v Hadoopu změnila a k čemu soubory slouží.

```
cd ~/hadoop-stack/hadoop-1.2.1/conf/
patch < ~/hadoop-stack/myhadoop/myhadoop-1.2.1.patch

cd ~/hadoop-stack/myhadoop/bin
sed -i 's/^print_nodelist | awk/cat $PBS_NODEFILE > $HADOOP_CONF_DIR\/slaves #/
    g' myhadoop-configure.sh
```

Výpis 10: Úprava Hadoopu dle myHadoop šablony

Pomocí knihovny myHadoop byly provedeny úpravy v následujících souborech:

- **Core-site.xml** - Soubor obsahuje definici složky pro umíštění dočasných souborů a adresu s portem kde běží NameNode čili hlavní služba pro HDFS.
- **Mapred-site.xml** - Soubor se skládá z URL adresy, na které poběží JobTracker, uzel rozdělující práci mezi podřízené výpočetní uzly.
- **Hdfs-site.xml** - Zde se nachází cesta k adresářům kde se budou ukládat data HDFS. Dále se pak nastavuje počet replikací dat na uzlech. V poslední části si můžete všimnout, že se zde opět nastavuje adresa na NameNode.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>HADOOP_TMP_DIR</value>
  </property>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://MASTER_NODE:54310</value>
  </property>
</configuration>
```

Výpis 11: Core-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property>
     <name>mapred.job.tracker</name>
     <value>MASTER_NODE:54311</value>
  </property>
  <property>
     <name>mapred.local.dir</name>
     <value>MAPRED_LOCAL_DIR</value>
     <final>true</final>
  </property>
  <property>
     <name>mapred.tasktracker.map.tasks.maximum</name>
     <value>MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM</value>
  </property>
  <property>
     <name>mapred.tasktracker.reduce.tasks.maximum</name>
     <value>MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM</value>
  </property>
  <property>
     <name>mapred.map.tasks</name>
     <value>MAPRED_MAP_TASKS</value>
  </property>
  <property>
     <name>mapred.reduce.tasks</name>
     <value>MAPRED_REDUCE_TASKS</value>
  </property>
</configuration>
```

Výpis 12: Mapred-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property>
     <name>dfs.name.dir</name>
     <value>DFS_NAME_DIR</value>
     <final>true</final>
  </property>
  <property>
```

```
    <name>dfs.data.dir</name>
    <value>DFS_DATA_DIR</value>
    <final>true</final>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>DFS_REPLICATION</value>
  </property>
  <property>
    <name>dfs.block.size</name>
    <value>DFS_BLOCK_SIZE</value>
  </property>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://MASTER_NODE:54310</value>
  </property>
</configuration>
```

Výpis 13: Hdfs-site.xml

### Nastavení systémových proměnných

V poslední kroku konfigurace pouze nastavíme systémové proměnné do souboru *bash_profile* dle Výpisu 14, aby jsme mohli jednoduše volat Hadoop pro spouštění úloh.

```
cd ~
echo "export JAVA_HOME=~/hadoop-stack/jdk1.7.0_79/" >> .bash_profile
echo "export HADOOP_HOME=~/hadoop-stack/hadoop-1.2.1/" >> .bash_profile
echo "export HADOOP_CONF_DIR=~/hadoop-stack/mycluster-conf-\$PBS_JOBID" >> .
    bash_profile
echo "export MH_SCRATCH_DIR=/lscratch/\$PBS_JOBID" >> .bash_profile
source ~/.bash_profile
```

Výpis 14: Nastavení systémových proměnných pro Hadoop

Kromě nastavení cesty k Javě a Hadoop, zde vidíme nastavení systémové proměnné MH_SCRATCH_DIR uchovávající cestu k lokálnímu úložišti daného uzlu.
Nastavením systémových proměnných je konfigurace dokončena.

### 3.6.2 Spouštění ukázkové úlohy

Každá verze Hadoopu obsahuje základní sadu programů vhodných na otestování správné konfigurace. V našem případě použijeme čítač slov v angličtině nazvaný WordCount. Před spouštěním úlohy musíme provést následující kroky: zažádat superpočítač o přiřazení výpočetních uzlů, spustit myHadoop, naformátovat distribuované úložiště do formátu HDFS a nakonec provést kopírování vstupního souboru do tohoto úložiště.

#### Žádost o výpočetní uzly

Pokud chceme provést na superpočítači nějaký složitý výpočet, musíme nejdříve zažádat o přidělení výpočetního výkonu. Superpočítač neobsluhuje požadavky hned, ale požadavky si ukládá do fronty, kde podle času přidání a priority úlohy postupně provádí jejich vykonání. Podrobnější popis naleznete v kapitole 2.4 Plánování spouštění úloh.

Přidělení čtyř výpočetních uzlů s 16 procesory v expresní frontě na maximálně hodinu výpočetního času provedete pomocí následujícího příkazu:

```
qsub -q qexp -l select=4:ncpus=16:walltime=1:00:00
```

Výpis 15: Přidělení výpočetního času pro Hadoop

#### Spouštení myHadoop

Po přidělení výpočetního výkonu budete automaticky přepnuti do konzole hlavního výpočetního uzlu. Dalším krokem je nastartování předkonfigurovaného Hadoop na hlavním uzlu a následně na všech podřízených. Spouštění provedete pomocí následující sekvence příkazů:

```
~/hadoop-stack/myhadoop/bin/myhadoop-configure.sh -c $HADOOP_CONF_DIR -s /
    lscratch/$PBS_JOBID/cag0008/
~/hadoop-stack/hadoop-1.2.1/bin/start-all.sh
```

Výpis 16: Spouštení Hadoop na Anselmu

V konzoli byste měli vidět výpis podobný Výpisu číslo 17. Z výpisu je patrné, že Hadoop při své inicializaci rovněž naformátoval distribuované diskové úložiště do formátu HDFS a spustil se jak na hlavním výpočetním uzlu, tak na podřízených výpočetních uzlech.

```
myHadoop: Keeping HADOOP_HOME=/home/cag0008/hadoop-stack/hadoop-1.2.1/ from
    user environment
myHadoop: Keeping MH_SCRATCH_DIR=/lscratch/328076.dm2 from user environment
myHadoop: Keeping HADOOP_CONF_DIR=/home/cag0008/hadoop-stack/mycluster-conf
    -328076.dm2
 from user environment
```

```
myHadoop: Using HADOOP_HOME=/home/cag0008/hadoop-stack/hadoop-1.2.1/
myHadoop: Using MH_SCRATCH_DIR=/lscratch/328076.dm2
myHadoop: Using JAVA_HOME=/apps/compilers/java/jdk1.7.0_21
myHadoop: Generating Hadoop configuration in directory in
 /home/cag0008/hadoop-stack/mycluster-conf-328076.dm2...
myHadoop: Using directory /scratch/cag0008/328076.dm2 for persisting HDFS state
    ...
myHadoop: Designating cn80.bullx as master node (namenode, secondary namenode,
    and
 jobtracker)
myHadoop: The following nodes will be slaves (datanode, tasktracer):
cn80.bullx
cn79.bullx
cn205.bullx
cn206.bullx
...
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = cn80/10.1.1.80
STARTUP_MSG: args = [-format, -nonIterative, -force]
STARTUP_MSG: version 1.2.1
/lscratch/328076.dm2/namenode_data has been successfully formatted.
17/05/19 09:45:37 INFO namenode.NameNode: SHUTDOWN_MSG:
```

Výpis 17: Výpis konzole při spouštění Hadoopu

Ověření správného nastartování Hadoopu lze provést velmi jednoduše pomocí příkazu z Výpisu 18.

```
~/hadoop-stack/hadoop-1.2.1/bin/hadoop dfsadmin -report
```

Výpis 18: Příkaz pro ověření nasazení Hadoopu

Pokud v konzoli vidíte uvidíte výpis podobný Výpisu 19, máte správně nastartovaný Hadoop připravený pro předložení první úlohy.

```
Configured Capacity: 1413179392000 (1.29 TB)
Present Capacity: 1341086502957 (1.22 TB)
DFS Remaining: 1341086388224 (1.22 TB)
DFS Used: 114733 (112.04 KB)
DFS Used%: 0%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
```

```
Datanodes available: 4 (4 total, 0 dead)
```

Výpis 19: Výpis ověření nasazení Hadoop

### Nakopírování souboru do HDFS

Hadoop očekává vstupní data ve svém distribuovaném souborovém systému HDFS, proto zde nahrajeme naše vstupní data před začátkem výpočtu.Se souborovým systémem HDFS se pracuje velmi podobně jako se soubory v příkazové řádce operačního systému Linux. Pro příklad zde je pár základních příkazů:

- **Výpis obsahu složky:** *bin/hadoop fs -ls*
- **Výpis obsahu souboru** *bin/hadoop fs -cat*
- **Vytvoření složky** *bin/hadoop fs -mkdir*
- **Kopírování souborů do HDFS:** *bin/hadoop fs -put*
- **Smazání složky nebo souboru** *bin/hadoop fs -rm*
- **Rekurzivní smazání** *bin/hadoop fs -rmr*
- **Výpis začátku souboru** *bin/hadoop fs -head*
- **Výpis konce souboru** *bin/hadoop fs -tail*
- **Nastavení práv k souboru či složce** *bin/hadoop fs -chmod*

Pro náš případ vytvoříme složku *input* do níž vložíte soubor *input.txt* s generovaným textem dle Výpisu číslo 20.

```
echo "hello world hadoop" >> input.txt
alias hadoop='~/hadoop-stack/hadoop-1.2.1/bin/hadoop'
hadoop fs -rmr input
hadoop fs -mkdir input
hadoop fs -put input.txt input/input.txt
```

Výpis 20: Nahrátí souboru do HDFS

### Spouštění úlohy

Nyní máme všechno připraveno a můžete se pustit do spouštění ukázkové úlohy čítač slov obsažené v každé verzi Hadoopu. Spouštění provedeme příkazy z výpisu 21.

```
cd ~/hadoop-stack/hadoop-1.2.1/bin/hadoop
hadoop jar hadoop-examples-1.2.1.jar wordcount input output
```

Výpis 21: Spouštění úlohy Hadoop

Po skončení výpočtu si můžete výsledek úlohy ověřit výpisem výstupního souboru ze složky *output* umístěného na distribuovaném disku. Ověření provedete příkazem z výpisu 22.

```
hadoop fs -cat output/part-r-00000
```

Výpis 22: Ověření výsledku Hadoop

Následně by se měl v konzoli vypsat text stejný s Výpisem číslo 23. Tímto krokem jste spustili první úlohu v Hadoopu na superpočítače a zároveň ověřili správnost konfigurace popsané v předchozí kapitole.

```
hello 1
word 1
hadoop 1
```

Výpis 23: Výsledek čítače slov Hadoop

# 4 Technologie Spark

V této kapitole je vysvětleno co je technologie Spark a krátký úvod do jeho historie. Poté následuje sekce popisující jeho vlastnosti v porovnání s předchozí zmíněnou technologií Hadoop. V další sekci je rozebrána architektura technologie Spark. Teoretickou část zakončuje popis distribuovaných datových sad zvaných RDD[19]. Poslední sekce vysvětluje nasazení Sparku na superpočítač Salomon.

### Definice

Spark je open source framework, napsaný v programovacím jazyku Java, určený pro efektivní paralelní zpracování velkého množství dat. Framework staví na programovém modelu MapReduce jako Hadoop a dále tento model rozšiřuje. To umožňuje daleko širší uplatnění Sparku pro nejenom dávkové zpracování dat, ale také interaktivní aplikace či SQL dotazování[4].

## 4.1 Historie

Spark byl jedním z Hadoop podprojektů vyvíjený od roku 2009 na univerzitě UC Berkeley's AMPLab mladým vědcem Matei Zaharia. Od roku 2010 se jedná o open source pod BSD licencí. Následně roku 2013 převzala projekt společnost Apache a od roku 2015 k vývoji Sparku přispívá více než 1000 programátorů. Spark se stal jedním z nejvíce aktivních open source projektů pro zpracování velkých dat[10].

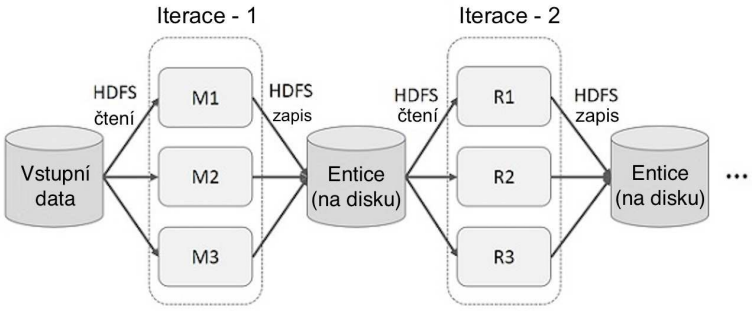
## 4.2 Vlastnosti

Spark překonává Hadoop v rychlosti zpracování dat, jednodušším používáním celého frameworku díky podpoře více programovacích jazyků a v neposlední řadě možností napojení na různé datové zdroje. Možná nevýhoda Sparku je absence vlastního souborového systému, musí spoléhat například na HDFS od Hadoopu. V následující sekci budou popsány jednotlivé vlastnosti podrobněji.

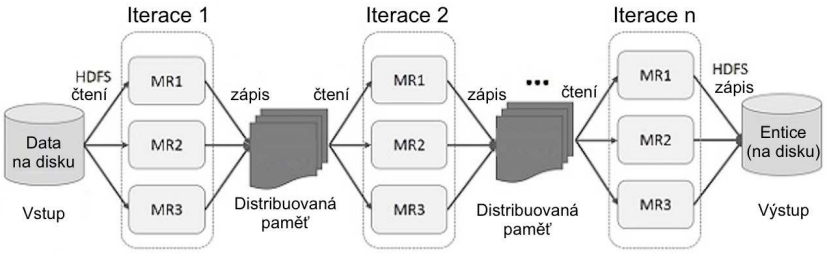
### Rychlost

Spark je velmi rychlý a překonává technologii Hadoop až 100x dle tvrzení firmy Apache[10], díky ukládání mezivýsledků části Map a Reduce do operační paměti místo na pevný disk. Technologie Hadoop pří každé operaci ať už Map nebo Reduce ukládá mezivýsledek na pevný disk. Ukládání mezivýsledku na disk celý proces výpočtu zpomaluje kvůli replikaci dat, serializaci dat z operační paměti na disk počítače a následně rychlosti IO operací disku. Zpracování iterativní úlohy Hadoopem je zobrazeno na Obrázku 9 a pro porovnání na Obrázku číslo 10 je zobrazeno zpracování úlohy Sparkem.

[19] Dokumentace k distribuovaným datovým sadám RDD: https://spark.apache.org/docs/latest/rdd-programming-guide.html

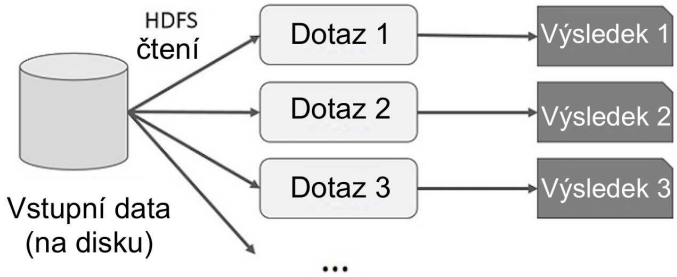



Obrázek 9: Iterativní zpracování v Hadoopu[4]

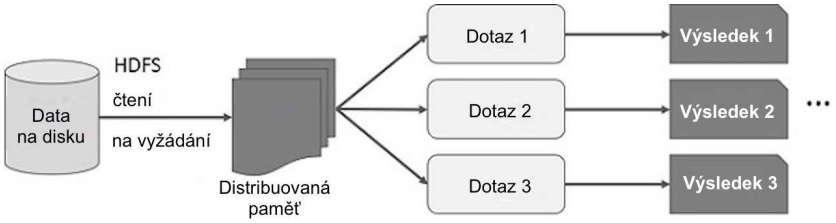



Obrázek 10: Iterativní zpracování ve Sparku[4]

Další z věcí kde Hadoop zaostává jsou interaktivní operace nad MapReduce algoritmem. Pokud uživatel bude volat několikrát za sebou různé dotazy nad stejnými daty, Hadoop vždy čte data znova z pevného disku. Naproti tomu Spark dokáže rozpoznat pokud voláme rozdílné dotazy na stejná data a uložit si data do operační paměti a tím zrychlit dotazování[4]. Pro zpracování mezivýsledků v operační paměti se používají RDD datasety popsané v pozdější části kapitoly. Porovnání zpracování interaktivního volání dotazů na stejná data Hadoopem můžete vidět na Obrázku 11 a Sparkem na Obrázku 12.




Obrázek 11: Interaktivní zpracování v Hadoopu[4]




Obrázek 12: Interaktivní zpracování ve Sparku[4]

### Jednoduchost používání

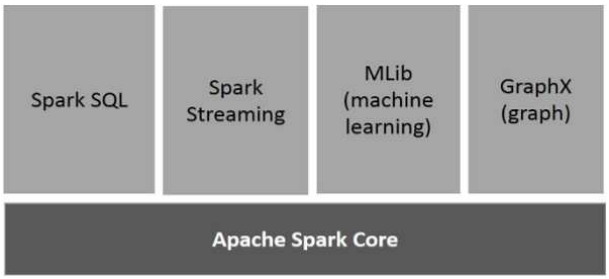
Spark podporuje přes 80 vysokoúrovňových operací umožňující jednoduché vytváření paralelních aplikací. Jako například map, flatMap, filter, union, intersection, sample, atd. Kromě toho dává programátorovi větší volnost při psaní kódu, aktuálně si programátor může vybrat mezi třemi podporovanými jazyky jako je Java, Python a Scala. Velkou výhodou při testování algoritmů je možnost provádění interaktivního volání výše zmíněných operací v Spark konzoli.

### Obecnost

Poslední z jeho významných vlastností je obecnost. Spark může běžet kdekoli ať už připojený k Hadoop ekosystému, Mesosu[20] nebo sám jako jediná instance. Mesos je program zpřístupňující zdroje z datacentra (CPU, operační paměť, disk) a zajišťuje toleranci vůči chybám. Spark rovněž podporuje přístup k různým datovým zdrojům jako HDFS, Cassandra[21], HBase[22] a Amazon S3[23][10].

## 4.3 Architektura

Spark se skládá z pěti modulů, přičemž nejdůležitější modul je Apache Spark Core, nad kterým jsou postaveny ostatní moduly. Architekturu můžete vidět na Obrázku 13.




Obrázek 13: Moduly ve Sparku[4]

[20]Dokumentace k Mesos: http://mesos.apache.org
[21]Dokumentace k Cassandra Db: https://academy.datastax.com/resources/brief-introduction-apache-cassandra
[22]Dokumentace k HBase databází: https://hbase.apache.org/
[23]Introduction to Amazon S3: http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html

- **Apache Spark Core:** hlavní modul zajišťující rozdělení úlohy na menší části a následně je zodpovědný za jejich zpracování v distribuovaném prostředí několika výpočetních uzlů. Rovněž je zodpovědný za základní IO operace do operační paměti a pevného disku. Uvnitř modulu jsou implementované RDD datové kolekce, které budou podrobněji popsané v další kapitole. Modul pro komunikaci s okolním světem vystavuje rozhraní v programovacích jazycích Java, Python a Scala.

- **Spark SQL:** modul podporuje dotazování nad strukturovanými daty, například z relační databáze. Pro připojení k databázi lze použít ODBC nebo JDBC ovládač. Modul je postavený nad základním modulem Apache Spark Core. K získávání dat lze použít klasické SQL dotazy nebo využít API poskytované přmo Sparkem nazvané DataFrame[24].

- **Spark streaming:** modul určený pro proudovou analýzu dat, které jsou přijímána průběžně v malých dávkách. Modul umožňuje i nad těmito daty volat vysokoúrovňové operace z RDD kolekcí. Proudové data můžou být přijímána například z nástroje Kafka[25], Flume[26], Twitteru nebo TCP-IP soketů.

- **MLib:** modul obsahující statistické a machine learning algoritmy. Uvnitř modulu lze nalézt různé druhy algoritmů pro klasifikaci (regrese, Naive Bayes klasifikátor), redukci dimenze (SVD, PCA), shlukování (K-means) a mnohé další[27].

- **GraphX:** modul pro efektivní reprezentaci neměnných grafů ve Sparku. Pokud používáte často měnitelné grafy, je lepší použít přímo grafové databáze než framework Spark. Uvnitř modulu lze nalézt algoritmy jako nejkratší cesta v grafu, kostra grafu či PageRang ohodnocující důležitost webové stránky, například na základě počtu odkazů vedoucí na danou stránku[10].

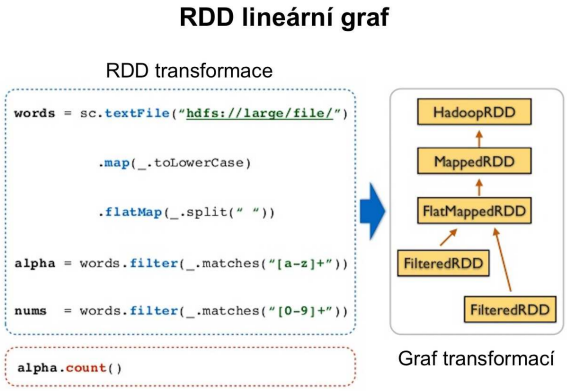

Obrázek 14: Frameworky spolupracující se Sparkem[10]

[24]Ukázka práce s DataFreme: https://docs.gigaspaces.com/xap/12.3/dev-java/insightedge-dataframes.html
[25]Dokumentac Kafka: https://kafka.apache.org/intro
[26]Dokumentace Flume: https://flume.apache.org/documentation.html
[27]Podrobnější seznámení s machine learning algoritmy naleznete na: https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/

## 4.4 RRD (Resilient Distributed Dataset)

Základní datová struktura ve Sparku představující neměnnou distribuovanou kolekci objektů. Každá taková struktura je rozdělena do několika logických částí, které můžou být počítány na různých výpočetních uzlech. Struktura uvnitř může obsahovat libovolné typy z programovacích jazyků Java, Pyhon nebo Scala. Rovněž struktura podporuje umístění uživatelsky definovaných objektů.[4]

Strukturu můžeme vytvořit dvěma způsoby, první z nich je při transformaci nad existující strukturou vracející novou strukturu a druhý způsob je vytvořením odkazu na externí datové úložiště jako HDFS, HBase nebo jiné podporované. Struktury jsou označované jako "lazy", což znamená vyhodnocení operací nad strukturou až je opravdu třeba, nikoli hned při definici operaci nad strukturou. Jedná se o výkonnostní optimalizaci, kdy se dopředu nevyhodnocuje nic, co nevíme jestli bude třeba v dalších krocích programu. Návrháři Sparku přemýšleli i o odolnosti vůči chybám způsobenými odpojením některého z výpočetních uzlů obsahující část datové struktury. Ke každé datové struktuře je ukládán lineární graf operací prováděných nad danou strukturou. Graf umožňuje zpětně provést operace a navíc šetří místo oproti technice ukládání záložních kopií částí struktur na různé výpočetní uzly. Uložení historie operací do lineárního grafu je zachyceno na Obrázku 15.




Obrázek 15: Linerní graf nad RDD strukturou[10]

Speciálním typem datových struktur jsou broadcast RDD struktury, vyznačující se vlastností automatického rozeslání na všechny připojené výpočetní uzly. Jejich použití je vhodné pouze ve speciálních případech, kdy výpočetní úloha potřebuje mít na každém výpočetním uzlu stejná data pro další počítaní. Spark se snaží co možná nejefektivněji přeposlat strukturu na výpočetní uzly, ale stále se řeší operace serializace a následné deserializace dat představující snížení výkonu aplikace. Proto není vhodné používat broadcast struktury moc často v aplikaci[10].

## 4.5 Nasazení Spark

V této sekci bude popsané nasazení technologie Spark na superpočítač Salomon, protože ve výchozím stavu má v dostupných modulech Spark framework. Stačí pouze správně nakonfigurovat hlavní a pomocné výpočetní uzly před předložením úlohy. Níže zmíněný postup vyžaduje, aby čtenář znal operační systém Linux minimálně na úrovni práce s Bash terminálem, protože celý postup nasazení probíhá přes terminálovou konzoli přistupující vzdáleně na superpočítač.

Poznámka: Ke konci diplomové práce se podařilo domluvit s týmem zajišťující podporu superpočítačů, aby doinstalovaly Spark do dostupných modulů i na superpočítač Anselm. Takže návod platí pro oba superpočítače.

### Přihlášení k superpočítače

Pro přihlášení k superpočítači se používá SSH protokol, pokud používáte Linux zadejte do příkazové řádky příkaz z Výpisu 24 a následně budete vyzvání k zadání uživatelského jména a hesla. Po zadání se vzdáleně přihlásíte pod svým účtem do superpočítače a můžete zadávat příkazy. Pokud používáte operační systém Windows na svém klientské počítači můžete pro přihlášení použít program WinSCP nebo Putty. K přihlášení je třeba vygenerovaný privátní klíč. Podrobný návod jak získat přístup a vygenerovat klíč můžete nalézt v dokumentaci IT4Innovations[28].

```
ssh -i id_rsa yourUsername@salomon.it4i.cz
```

Výpis 24: Přihlášení do superpočítače

Po úspěšném přihlášení byste měli vidět zprávu v terminálu podobné této:


Obrázek 16: Zpráva po přihlášení Salomon

[28] Generování klíče: https://docs.it4i.cz/anselm/shell-and-data-access/

### 4.5.1 Příprava prostředí

V následující kapitole je popsáno jak vytvořit strukturu, složku a nasazovací skripty aplikace, aby se mohla aplikace jednoduše spustit na počítači Salomon.

#### Generování šablony

Před použitím Sparku je nutné nejdříve si vytvořit složku, nazveme si ji třeba *spark_example_task*, obsahující podsložku *jobs* a tři Bash skripty pojmenované *run-spark-jobs.sh, salomon-start-slave.sh a clean-project.sh*. K skriptům je třeba ještě dodatečně nastavit práva na spouštění. Skripty můžete vytvořit a nastavit pomoci následujících příkazů zobrazených ve Výpisu 25.

```
mkdir spark_example_task
cd spark_example_task
mkdir jobs
touch run-spark-jobs.sh
touch salomon-start-slave.sh
touch cleanProject.sh
chmod +x run-spark-jobs.sh
chmod +x salomon-start-slave.sh
chmod +x clean-project.sh
```

Výpis 25: Proces vytvoření struktury složky Spark

#### Tvorba skriptu run-spark-jobs.sh

Ve skriptu *run-spark-jobs.sh* se nejdříve musí načíst potřebné moduly. Modul s jazykem Python je automaticky načten po přihlášení k superpočítači a není třeba jej načítat, jediný potřebný modul pro načtení je Spark. V době psaní diplomové práce byla na Salomonu verze Sparku 2.1.2. Poté ve skriptu exportujeme cestu k Javě a zastavíme všechny běžící Java procesy, které mohou představovat běhy Sparku z předchozí úlohy. Následně se vytvoří URL na níž poběží Spark, složená z názvu počítače a portu 7077. Nejprve se spustí Spark na hlavním uzlu označený jako master provádějící příjem úloh. Poté se z systémové proměnné PBS_NODEFILE získají jména dalších alokovaných výpočetních uzlů, které se pro Spark nastaví jako podřízené(Slave). Nakonec se provádí skenování složky jobs na možné programy, které se předloží Sparku k výpočtu. Celý skript je zobrazen na další straně ve Výpisu 26.

```
#!/bin/sh
module load Spark
export JAVA_HOME

killall java && sleep 4

PORT=${SPARK_MASTER_PORT:-7077}
URL=spark://`hostname`:${PORT}

echo $URL

# Start master
start-master.sh || exit 1

# Start workers
for server in `cat $PBS_NODEFILE` ; do
 ssh ${server} $SPATH/salomon-start-slave.sh ${URL} ${SPARK_WORKER_DIR} ${
     SPARK_LOG_DIR} || exit 1
done

for i in $SPATH/jobs/.*.py ; do
 echo Submitting job $i
 spark-submit --master ${URL} ${i}
done
```

Výpis 26: Skript run-spark-jobs

### Tvorba skriptu salomon-start-slave.sh

Skript *salomon-start-slave.sh* provádějící inicializaci podřízených výpočetních uzlů, skript je volán automaticky po inicializaci hlavního výpočetního uzlu skriptem *run-spark-jobs.sh*. Ve skriptu se nejdříve načte modul Spark, poté se nastaví cesty pro pracovní a logovací složku a nakonec se předá URL na které bude uzel běžet. Celý skript je zobrazen ve Výpisu 27.

```
#!/bin/sh

if [ "$#" -ne 3 ]; then
   echo "Usage: $0 <url> <worker-dir> <log-dir>"
   exit 1
```

```
fi

source /etc/profile
module load Spark || exit 1

URL=$1
export SPARK_WORKER_DIR=$2
export SPARK_LOG_DIR=$3

start-slave.sh ${URL} &
#wait
```

Výpis 27: Skript salomo-start-slave.sh

#### Tvorba skriptu clean-project.sh

Poslední skript *clean-project.sh* slouží pro vyčištění projektu, pokud chceme znovu spustit úlohu. Smaže složku s výsledky, metadaty a logy. Obsah je ukázán ve výpisu 28.

```
rm -rf result
rm -rf metastore_db/
rm -rf spark-logs
rm -f derby.log
```

Výpis 28: Skript clean-project.sh

### 4.5.2 Spouštění ukázkové úlohy

V této kapitole bude ukázáno jak spustit jednoduchý algoritmus počítání počtu slov v textovém souboru za použití frameworku Spark. Celý proces se skládá ze tří kroků, první je zažádání o výpočetní čas na superpočítači, poté napsaní Python skriptu obsahující zmíněný algoritmus ve složce *jobs* a nakonec spouštění skriptu *run-spark-jobs.sh*.

#### Žádost o výpočetní uzly

Pokud chceme provést na superpočítači výpočet, musíme nejdříve zažádat o přidělení výpočetního výkonu. Superpočítač neobsluhuje požadavky ihned, ale ukládá si je do fronty. Následně podle času přidání a priority úlohy superpočítač vyzvedává úlohy z fronty a provádí jejich zpracování. Podrobnější popis naleznete v kapitole 2.4 Plánování q spouštění úloh.

Přidělení 4 výpočetních uzlů s 16 procesory v expresní frontě na maximálně hodinu výpočetního provedete pomocí příkazu z Výpisu 29.

```
qsub -q qexp -l select=4:ncpus=16:walltime=1:00:00
```

Výpis 29: Přidělení výpočetního času WordCount

### Napsání programu čítač slov

Už bylo zmíněno jak vytvořit struktuturu, teď vytvoříte ve složce *jobs* soubor *program.py* obsahující kód algoritmu čítač slov, který se bude předkládat Sparku. Kompletní ukázku algoritmu můžete nalézt v příloze na konci práce, zde ve výpisu 30 je uveden jenom náhled. Důležité je nastavit správně proměnnou *spark context* na URL kde běží Spark a cesty k souborům.

```
inputFile="input.txt";
outputDirectory="result";
sc = SparkContext("spark://"+hostname+":7077") # HPC clusters
lines = sc.textFile(inputFile);
countWords = lines.flatMap(lambda line: line.split(" ")) \
        .filter(lambda word: isWord(word)) \
        .map(lambda word: stem(word.lower())) \
   .map(lambda word: (word, 1)) \
   .reduceByKey(lambda x, y: x + y ) \
   .sortByKey(True);
countWords.saveAsTextFile(outputDirectory);
```

Výpis 30: Ukázka algoritmu čítač slov ve Sparku

Nakonec vytvořte ve složce *spark_example_task* soubor *input.txt* s textem "hello world spark", potřebný pro ukázkové spouštění úlohy.

### Spouštění úlohy

Předložení úlohy Sparku se provede velmi jednoduše, stačí, když spustíte skript *run-spark-jobs.sh*. Skript následně provede spouštění Sparku na všech výpočetních uzlech a předloží úlohu ze složky *jobs*. Po výpočtu byste měli vidět ve složce *result* soubor s níže zmíněným obsahem, a tím se vám podařilo správně nakonfigurovat Spark a spustit ukázkovou úlohu.

```
hello 1
word 1
spark 1
```

Výpis 31: Výsledek čítače slov Spark

# 5 Implementované algoritmy

V diplomové práci byly implementovány celkem tři algoritmy WordCount, Invertované indexování a K-means. V následujících kapitolách jsou popsány dva složitější z nich a to: Invertované indexování, představitel textové analýzy, a K-means zástupce machine learning algoritmů. U každého algoritmu je popsán obecný princip fungování a následně jak probíhá výpočet pomocí programového modelu MapReduce v distribuovaném prostředí. Algoritmy byly implementovány jak pro Hadoop tak Spark, porovnání zápisu algoritmů je v příloze na konci práce, kde jsou ukázky zdrojových kódu.

## 5.1 Invertované indexování

### Úvod

Webové vyhledávání je typický Big data problém[18]. Uživatel chce na Internetu najít nějaké informace a zadá do vyhledávače hledaný výraz. Očekává od vyhledávacího nástroje rychlou odpověď v řadě několika sekund. Kdybychom použili sekvenční průchod přes webové stránky dotaz by trval neúměrně dlouho a uživatel by ztratil trpělivost čekat na výsledek. Proto se pro účely vyhledávání webových stránek používá Invertované indexování, které je rovněž základem všech fulltextových vyhledávačů[18]. Technika Invertovaného indexování představuje datovou strukturu ukládající obsah do mapy. Pro daný výraz je uložen seznam dokumentů (webových stránek) obsahující dané slovo. Invertovaného indexování se nezajímá o to na kterém místě v dokumentu se slovo nachází, značí si pouze název dokumentu, kde se slovo vyskytuje. Při vyhledávání nás totiž nezajímá, jestli slovo je na prvním nebo desátém řádku, ale zajímá nás, jestli se dané slovo v dokumentu vyskytuje a popřípadě kolikrát. Počet výskytu slova v dokumentu značí, jak moc článek vystihuje právě vyhledávaný výraz[17].

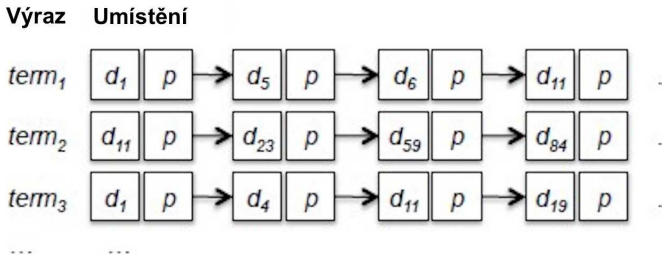
$$
\begin{aligned}
&\text{výraz}_1 \text{ se vyskytuje v } \{ \text{dokumentu}_1, \text{dokumentu}_5, \text{dokumentu}_{11}, \ldots\},\\
&\text{výraz}_2 \text{ se vyskytuje v } \{ \text{dokumentu}_{11}, \text{dokumentu}_{23}, \text{dokumentu}_{84}, \ldots\},\\
&\text{výraz}_3 \text{ se vyskytuje v } \{ \text{dokumentu}_1, \text{dokumentu}_4, \text{dokumentu}_{11}, \ldots\},
\end{aligned}
$$

*Technika Invertovaného indexování dokumentů nám říká, ve kterých dokumentech se nachází hledaný výraz X[11].*

### Popis algoritmu

Invertované indexování slouží pro rychlé full-textové vyhledávání. Algoritmus využívá mapu k uložení odkazů na vyhledávané termíny. Klíč je termín a jako hodnota seznam dokumentů, kde se daný termín vyskytuje. Seznam dokumentů bývá často rozšířen ještě o další informace, jako

například kolikrát se daný termín v dokumentu vyskytuje. V některých případech potřebujeme vědět i váhu slova, tak se seznam dokumentů rozšíří o další atribut, signalizující zda se termín vyskytuje v nadpisu dokumentu nebo až v textu[11]. Na Obrázku 17 můžete vidět strukturu Invertovaného indexování.




Obrázek 17: Podrobný pohled na Invertovaného indexování[11]

### Ukázka

V následující ukázce byly použity na vstupu čtyři dokumenty zobrazené ve Výpisu 32 , každý dokument představuje jeden řádek označený číslem. Výstupem indexování je mapa ukázaná ve Výpisu 33, klíč představuje termín a jako hodnota je počet výskytů v jednotlivých dokumentech. Například slovo (termín) „if“ se vyskytuje v dokumentech čtyřikrát, což značí první číslice. Za první číslicí lze vidět pole entic. Každá entice reprezentuje výskyt slova v určitém dokumentu. První číslice v entici značí číslo dokumentu a druhá číslice vyjadřuje počet výskytů v daném dokumentu.

```
1: if you prick us do we not bleed
2: if you tickle us do we not laugh
3: if you poison us do we not die and
4: if you wrong us shall we not revenge
```

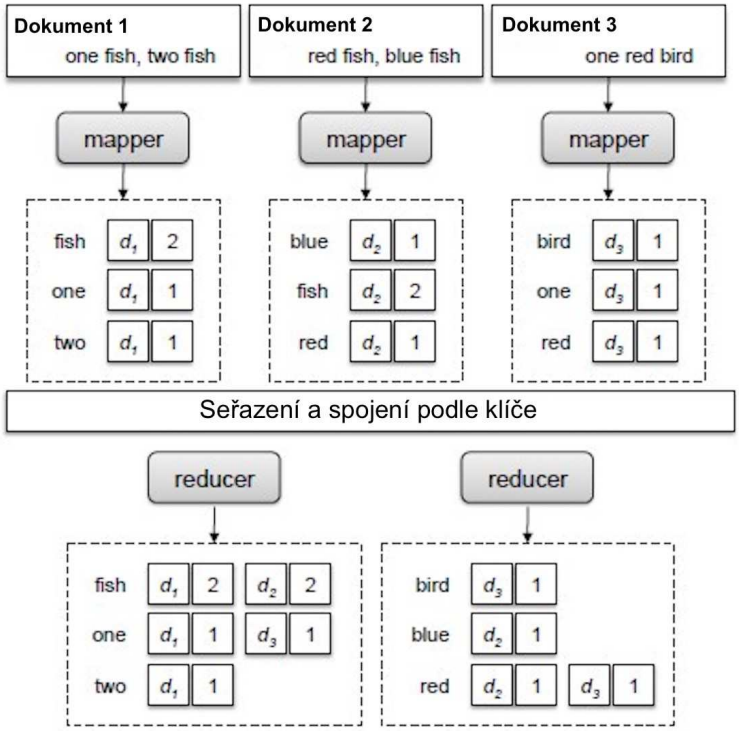
Výpis 32: Ukázkový vstup pro Invertované indexování

```
and     : 1 : (3, 1)
bleed   : 1 : (1, 1)
die     : 1 : (3, 1)
do      : 3 : (1, 1), (2, 1), (3, 1)
if      : 4 : (1, 1), (2, 1), (3, 1), (4, 1)
laugh   : 1 : (2, 1)
...
```

Výpis 33: Ukázkový výstup z Invertovaného indexování

### MapReduce provedení

Před začátkem algoritmu jsou jednotlivé dokumenty nebo jejich části rozdistribuovány mezi připojené výpočetní uzly. Poté se na jednotlivých výpočetních uzlech začne vykonávat funkce Map, která rozdělí řádek na jednotlivá slova a následně zapíše na výstup jako mapa klič-hodnota. Klíč představuje dané slovo a hodnota je datová struktura tvořící dvojici atributů číslo dokumentu a počet výskytů. Funkce Map si do atributu počet výskytů vloží vždy jedničku, což představuje jeden výskyt právě zpracovávaného slova. Po úspěšném zpracování všech vstupních dokumentů následuje funkce Combine, provádějící sjednocení stejných klíčů do jednoho s tím, že jejich hodnoty slouží do jednoho seznamu. Kroky výpočtu Invertovaného indexování můžete vidět na Obrázku číslo 18. V poslední části výpočtu se zavolá funkce Reduce načítající od každého klíče(slova) seznam výskytů a provádí součty výskytů slov pro jednotlivé dokumenty[17].




Obrázek 18: Průběh Invertovaného indexování pomocí MapReduce[17]

## 5.2 K-means

K-means je jedním z představitelů machine learning algoritmů, který patří do kategorie algoritmů bez učitele používaný ke shlukování. Cílem shlukování je v dané množině objektů nalézt její podmnožiny - shluky objektů - tak, aby si členové shluku byli navzájem podobní, ale nebyli si příliš podobní s objekty mimo tento shluk[20]. Algoritmus předpokládá, že shlukované objekty můžeme chápat jako body v eukleidovském prostoru a že počet shluků je pevně dán. Při analýze je vhodné vyzkoušet různý počet shluků a ověřit výsledky[11]. Algoritmus byl poprvé publikován Hugem Steinhausem v roce 1957, ale poprvé jej použil až James MacQueen v roce 1967. Dva roky před tím E. W. Forgy uveřejnil stejnou metodu, proto se někdy označuje také jako Lloydův-Forgyho algoritmus. Efektivní verze algoritmu byla naprogramována už v roce 1975-1979 pomocí programovacího jazyka Fortran[21].

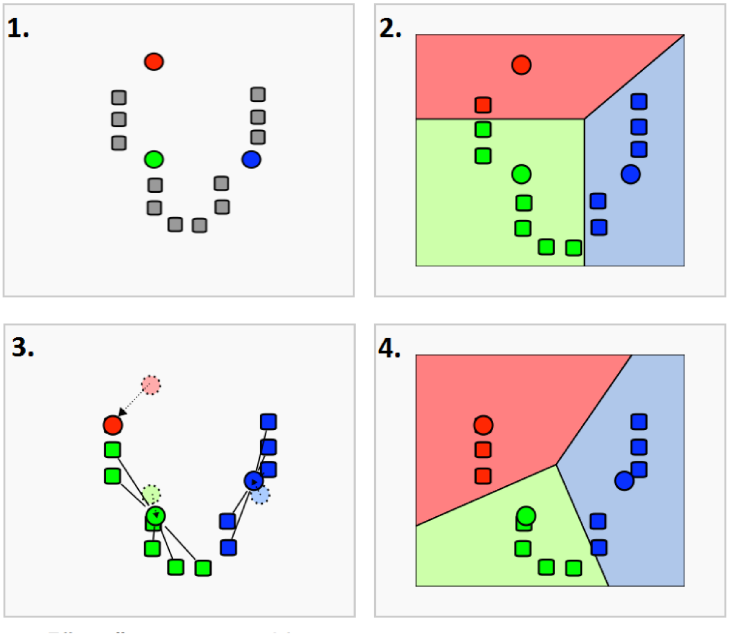
Algoritmus K-means se používá k nalezení shluků (skupin) v datech, které nemáme v datech vysloveně označené. To lze například využít v obchodě, kde chceme nalézt N skupin (shluků) zákazníků a podle toho uzpůsobit nabídku zboží v prodejně. Pokud máme záznamy všech nákupů jednotlivých lidí, s informací co si koupili a kolikrát, lze algoritmem K-means najít N skupin (shluků) nad sesbíranými daty. Uprostřed každého nalezené skupiny (shluku) leží centroid představující typického zákazníka pro danou skupinu. Dalším příkladem použití K-means může být doporučování článků čtenáři v internetových novinách, pokud si ukládáme jeho předchozí navštívené články[22].

### Popis algoritmu

Před začátkem výpočtu je třeba definovat kolik shluků má algoritmus najít, toto číslo označme jako N. Algoritmus začne tím, že si ze vstupní množiny objektů vybere právě N objektů a označí si je jako centroidy. Výběr může být nahodilý nebo pevně dán výběrem prvních N objektů z množiny. Poté se projdou všechny zbylé objekty a přiřadí se k nejbližšímu centroidu dle definované metriky. Metrika vyjadřuje podobnost mezi objekty[20]. K nejznámějším metrikách patří Euklidova, Kosinova nebo Manhattanská[24]. Jakmile algoritmus dokončí přiřazení objektů k centroidům, vypočte nové centroidy, tak aby šlo o těžiště shluků. Pak se celý proces přiřazení zbylých bodů k novým centroidům opakuje do doby, než se poloha centroidů ustálí nebo algoritmus projde maximální počet iterací definovaných na vstupu. Ustálení lze definovat tak, že se už nemění nově nalezené centroidy od předchozích a nebo je odchylka velmi malá. Algoritmus má zaručenou konvergenci, ale nevýhoda algoritmu může být vznik různých výsledků při jiném zvolení počátečních centroidů a také nemusí dojít k globálnímu optimu ve smyslu nejmenšího součtu čtverců vzdáleností od centroidů[11].

### Ukázka

Obrázek 19 vykresluje jednotlivé kroky zpracování algoritmu K-means pro nalezení tří shluků v skupině 15 objektů. V prvním kroku se vyberou náhodně tři objekty a označí se jako centroidy. Ve druhém kroku dojde k přiřazení zbylých bodů k nejbližším centroidům, čímž vzniknou tři shluky. U třetího kroku se vypočtou nové centroidy v nalezených shlucích a to tak, aby šlo o těžiště objektů patřících do těchto shluků. Následně se opakují kroky 2 a 3, dokud nedojde k ustálení centroidů nebo překročení maximálního počtu iterací.



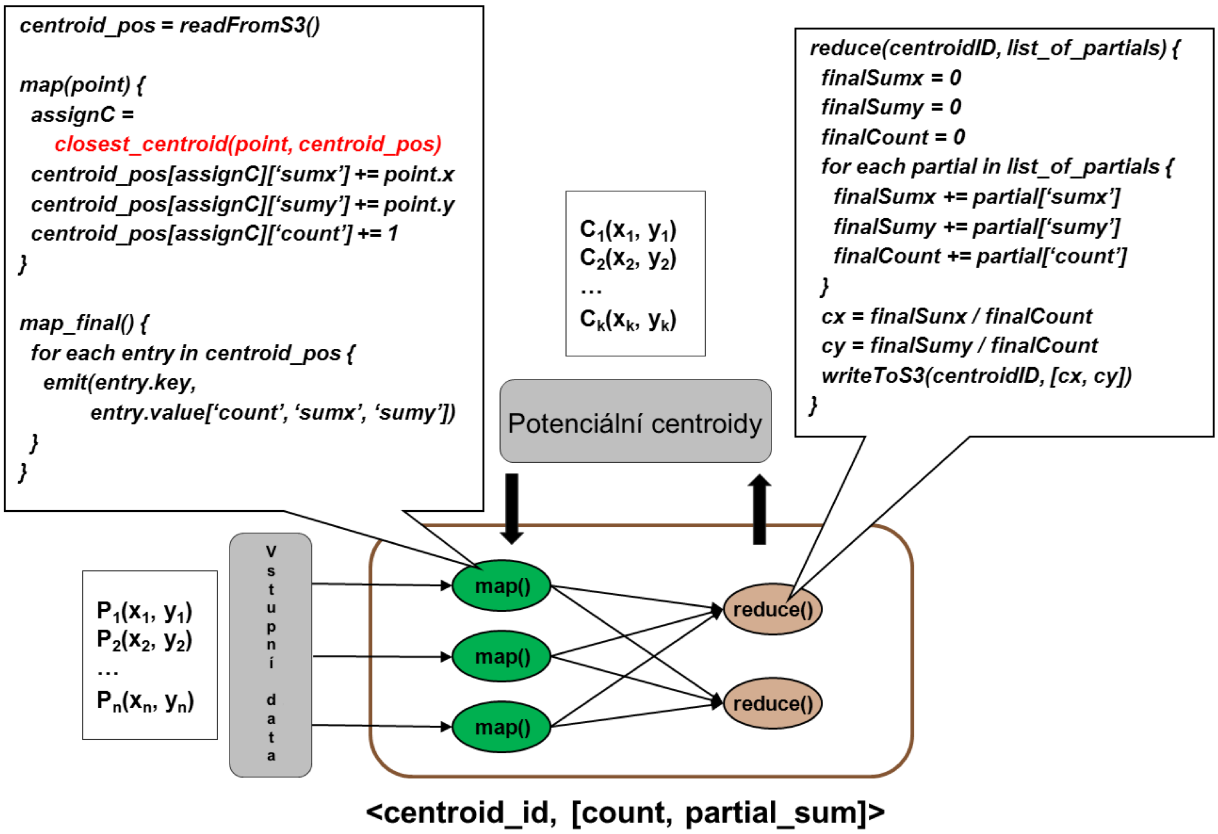

Obrázek 19: Grafická ukázka K-means[22]

### MapReduce provedení

Algoritmus K-means v programovém modelu MapReduce nejdříve načte všechny objekty ze vstupního souboru a provede přerozdělení objektů mezi jednotlivé výpočetní uzly. V dalším kroku se provede inicializace centroidů, algoritmus vezme N objektů z načteného souboru a označí si je jako centroidy. Označené centroidy se rozešlou mezi všechny výpočetní uzly jako neměnná globální proměnná.

Poté dojde k zavolání funkce Map, která provede přerozdělení objektů k nejbližším centroidům na každém výpočetním uzlu. Ještě než je zavolána funkce Reduce, využijí se data uložené na výpočetních uzlech a provede se funkce Combine. Tato funkce vezme objekty přiřazené k centroidu a uloží si místo nich počet objektů a součet objektů u daného centroidu, tím se šetří velikost přenášených dat mezi uzly. Funkce Reduce pak z před připravených výsledků funkce Combine provede spočtení celkového součtu a celkového počtu objektů pro daný shluk.

Po dokončení funkce Reduce je spočtený celkový počet objektů a celkový součet objektů pro daný shluk. Na základě této informace lze určit nový centroid v daném shluku, vydělením celkového součtu celkovým počtem objektů ve shluku. Následně pokud není splněna ukončovací podmínka, jako například ustálení polohy centroidů nebo překročení maximálního počtu iterací, tak se rozešlou nově spočtené centroidy jako globální proměnné mezi jednotlivé uzly a opakuje se volání funkcí Map, Combine a Reduce[19].



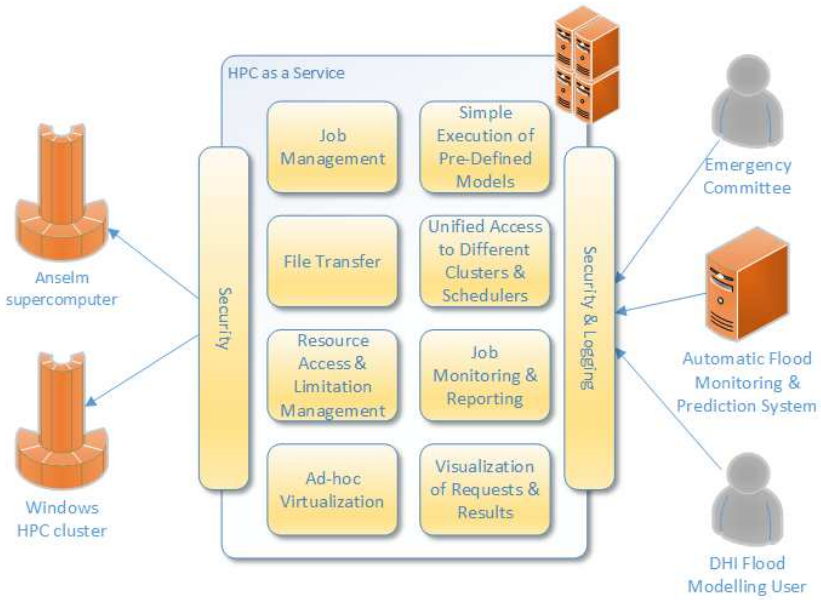

Obrázek 20: Průběh K-means pomocí MapReduce[25]

# 6 HPC as a Service

Kromě implementace algoritmů bylo součástí práce vytvoření grafického klienta umožňující implementované algoritmy spouštět vzdáleně z osobního počítače. Pro předložení úlohy na superpočítač a následné zpracování byla použita služba HPC as a Service, nad kterou je postaven implementovaný klient. Nejdříve je v této kapitole rozebráno jak funguje služba HPC as a Service, následně jak probíhá komunikace s touto službou a poté vlastní implementace klienta.

## 6.1 Popis služby

V Národním superpočítačovém centru IT4Innovations[29] vznikla služba HPC as a Service představující nadstavbu nad superpočítačem provádějící spouštění úloh, přidělování zdrojů a přenos souborů na jednotlivé výpočetní uzly. Celá služba je postavena na architektuře klient-server komunikující pomocí webového rozhraní SOAP jak lze vidět na Obrázku 21. Webové rozhraní zajišťuje nutnou funkcionalitu jako spouštění, plánování a monitorování úloh. Dále podporuje reporty, přenos dat na výpočetní uzly a zpětné stažení dat po dokončení úlohy z výpočetních uzlů. K webové službě mohou přistupovat pouze oprávnění uživatelé, proto je služba rozšířena o autorizaci a autentizaci[26].

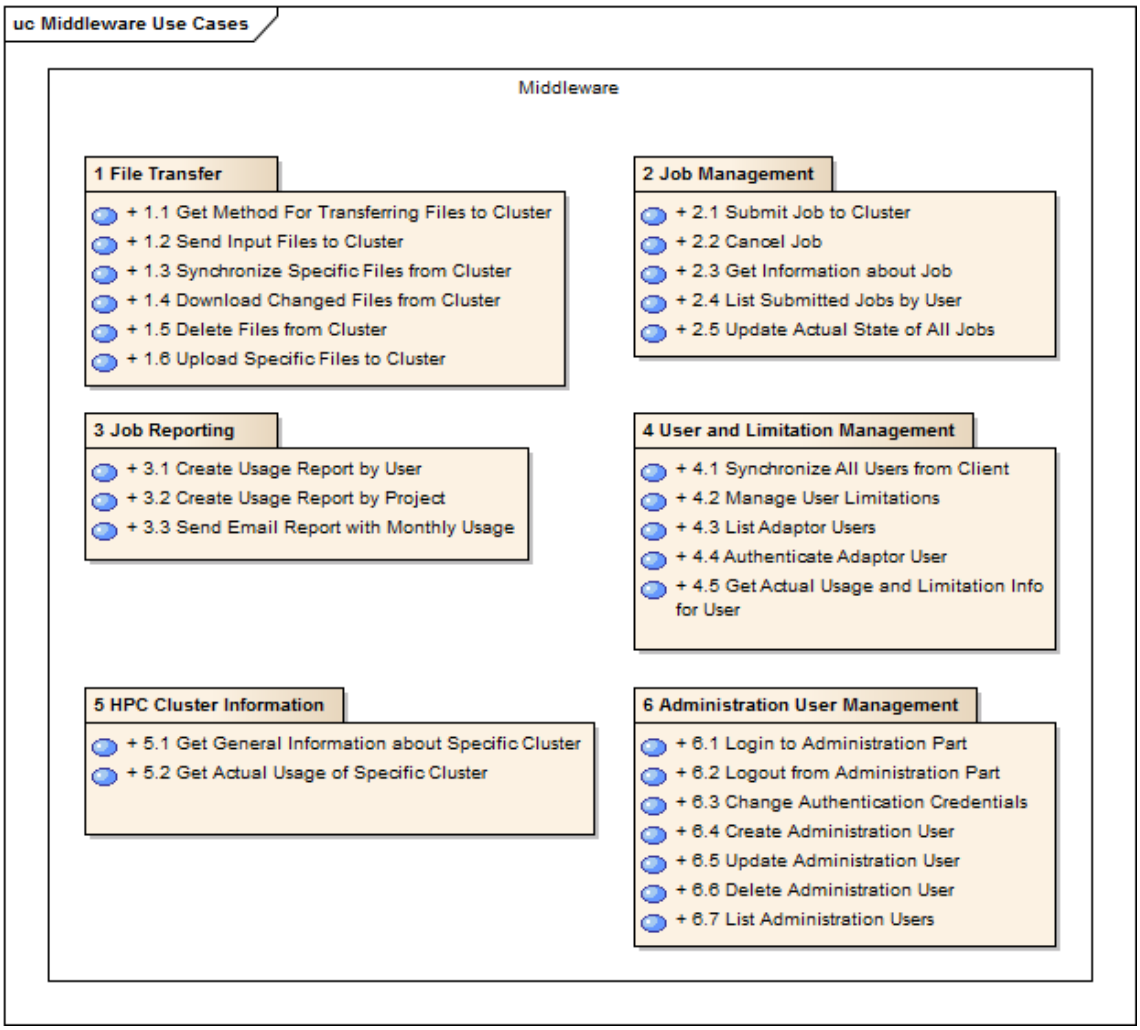



Obrázek 21: Vysokoúrovňový pohled na HPC as a Service[26]

[29] IT4Innovations: https://www.it4i.cz/

## 6.2 Architektura

Tato kapitola popisuje vysokoúrovňový návrh modulů služby HPC as a Service z pohledu uživatele. Služba je rozdělena do šesti logických modulů a podporuje spouštění úloh na obou superpočítačích: Salomonu a Anselmu.




Obrázek 22: Architektura HPC as a Service[26]

- **File Transfer**: modul zajišťuje funkce pro přenos souborů z klientské aplikace na výpočetní uzly v HPC prostředí a naopak po dokončení úlohy. Kromě přenosu podporuje možnost smazání souborů svázaných s úlohou.
- **Job Management**: modul obsahuje funkce pro správu uživatelských úloh spuštěných na superpočítači, mezi jeho klíčové funkce patří spouštění nové úlohy na výpočetních uzlech v HPC, zrušení běžící úlohy a získání aktuálních informací o běžící úloze.

- **Job Reporting**: modul zobrazuje informace o využití clusteru jednotlivými uživateli a projekty. Dále obsahuje podporu pro automatické přeposlání měsíční analýzy na vybrané emaily správcům systému.
- **User and Limitation Management**: modul zajišťuje autentizaci a autorizaci uživatelů včetně kontroly jejich omezení jako například maximální počet výpočetních zdrojů pro danou úlohu.
- **HPC Cluster Information**: modul podává obecné informace o superpočítači jako celkový počet výpočetních uzlů, vytížení procesorů a počet aktuálně volných výpočetních uzlů.
- **Administration User Management**: je skupina funkcí pro vytváření, modifikování a smazání uživatelských účtů přístupných jenom administrátorům systému[26].

## 6.3 Komunikace se službou

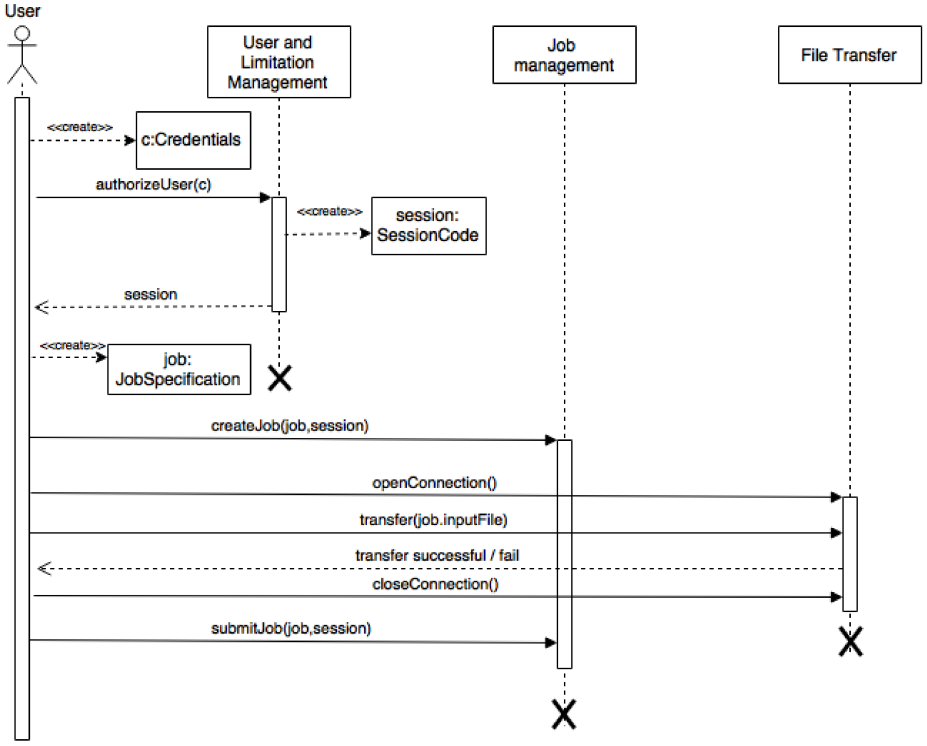
Vyvíjená aplikace komunikuje se službou HPC as a Service pomocí webové služby SOAP. Dvě nejvíce volané akce klientem, vyvíjeného v rámci diplomové práce, jsou předložení nové úlohy superpočítači a stažení výsledku úlohy, a proto budou v následující kapitole popsány. Obě aktivity komunikují s moduly User and Limitation Management, Job Management a File Transfer, jenž jsou součástí služby HPC as a Service.

### Předložení úlohy

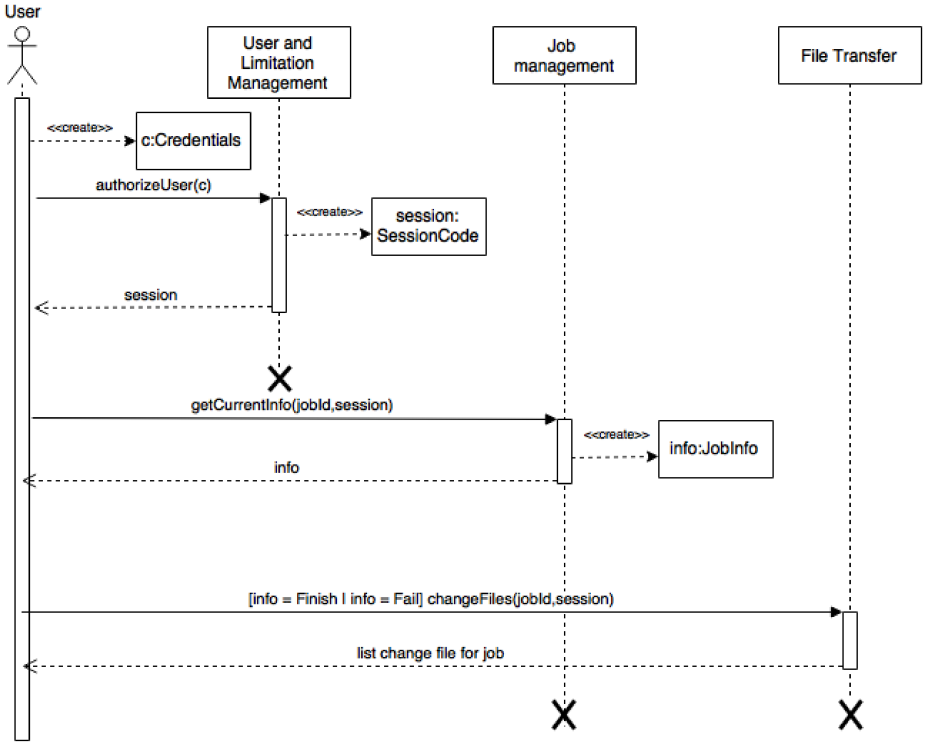
Před předložením úlohy se musí uživatel nejdříve přihlásit svými přihlašovacími údaji používanými pro přístup na HPC. Přihlášení zprostředkovává modul User and Limitation Management vracející token, který je nutný poslat při každé další komunikaci se službou pro ověření identity osoby. Následně uživatel vytvoří objekt obsahující všechny informace ohledně plánované úlohy, jako počet výpočetních uzlů, počet procesorů, čas pro běh, cestu k souborům. Objekt je poté předán metodou createJob modulu Job Management. Nahrání souborů na superpočítač zajišťuje modul File Transfer. Posledním krokem je zavolání metody submitJob na Job Management modulu. Celý proces je zachycen na Obrázku 23.

### Stažení výsledku úlohy

Před stahováním výsledku úlohy musí být uživatel přihlášený a znát přesné ID přidělené předložené úloze. Na základě ID se pomocí metody getCurrentInfo na modul Job Management získá stav předložené úlohy. Stavy mohou být čtyři: READY, COMPUTE, FINISH a ERROR. Pokud úloha proběhla v pořádku, je možné provolat File Transfer modul o poskytnutí změněných nebo nově vytvořených souborů. Celý proces je zachycen na Obrázku 24.




Obrázek 23: Předložení úlohy do HPC as a Service - sekvenční diagram




Obrázek 24: Stažení výsledku úlohy HPC as a Service - sekvenční diagram

## 6.4 Návrh klienta

### Úvod

Cílem bylo rozšíření rámec HPC as a Service o grafickou aplikaci umožňující vzdálené spouštění a monitorování implementovaných algoritmů v prostředí superpočítače. Jako superpočítač byl vybrán Salomon a tři testovací úlohy odpovídající implementovaným algoritmům pro technologii Spark v této práci. Hlavním cílem klienta je zjednodušení práce s superpočítačem pomocí grafického rozhraní, kdy uživatel pouze vybere algoritmus přidá k němu vstupní data a o zbytek se postará aplikace. Aplikace převezme požadavky od uživatele a přepošle je na službu HPC as Service, která provádí fyzickou alokaci výpočetních uzlů, konfiguraci technologie Spark a samotné spouštění úlohy.

### Hlavní požadavky

- Předložení úlohy na HPC
- Import vstupních dat do HPC
- Ověření stavu úlohy
- Stažení výsledků z HPC

### Vedlejší požadavky

- Real-time validace vstupních parametrů
- Full textové vyhledávání nad všemi položkami v tabulce předložených úloh
- Vícevláknový běh zlepšující odezvu aplikace
- Přívětivý vzhled - inspirace Material Design[30] od Googlu

### Použité technologie

Záměr bylo vytvořit multiplatformní aplikaci s podporou co možná největší škály operačních systému, proto byl zvolen programovací jazyk Java splňující přesně tento požadavek. Programy pro Javu jsou převedeny do speciálně upravené formy kódu zvané bytecode[31], jenž je následně spouštěn ve virtuálním stroji JVM. Virtuální stroj JVM je v dnešní době naprogramovaný pro Windows, Linux a MacOs. To je velký rozdíl oproti jazyku jako je například C, kde program je zkompilován přímo pro konkrétní počítač[27]. Předložené úlohy jsou prováděny na superpočítači, tudíž neklade klient velké nároky na výkon a můžeme oželet menší ztrátu rychlosti aplikace při použití mezivrstvy virtuálního stroje. Druhým důvodem použití Javy byla dobrá podpora

[30] Úvod do materialDesign: https://material.io/guidelines/material-design/introduction.html
[31] Java bytecode: https://www.beyondjava.net/blog/java-programmers-guide-java-byte-code/

tvorby grafických aplikací, už v základní verze Javy obsahuje tři knihovny AWT[27], Swing[27] a novou knihovnu JavaFX[27].

V začátcích prototypování se prováděl vývoj v grafické knihovně Swing, ale bylo zde naráženo na omezené možnosti stylování komponent a dnes již zastaralý způsob psaní UI vzhledu v kódu programovacího jazyka Java. Po skončení vytváření prototypů jsem se rozhodl celého klienta psát v technologii JavaFX, protože má daleko blíže webovému návrhu. Celé UI si může programátor vytvořit pomocí značkovacího jazyka vycházejícího z HTML a pro úpravu vzhledu využít CSS styly. Navíc už v základu dodržuje strukturu návrhového vzoru MVC[13], čímž zlepšuje orientaci v kódu.

Ke komunikaci s službou HPC as Service se v klientu používá knihovna Spring[32] obsahující řadu užitečných metod pro komunikaci nad SOAP protokolem. Velkým ulehčením je podpora generování kódu na základě WDSL popisu. Ke správě všech potřebných knihoven, verze a buildu aplikace byl u klienta využíván nástroj Maven[33] stahující všechny potřebné závislosti k běhu aplikace při její kompilaci.

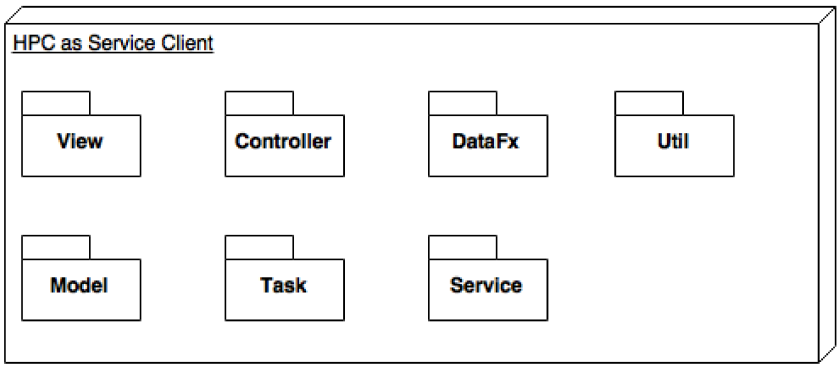
### Architektura

Z vysokoúrovňového pohledu se aplikace se opírá o návrhový vzor MVC(Model View Control). Grafická část komunikující s uživatelem představuje vrstvu View, zde je použita výše zmíněná technologie JavaFX. Jakmile uživatel vytvoří nějakou událost v prezenční vrstvě, je tato událost oznámená vrstvě Control provádějící validace vstupu od uživatele a zavolání příslušné logiky z vrstvy Model. Ve vrstvě model můžeme najít hlavní logiku aplikace komunikující se světem HPC.

Při pohledu přímo do kódu klienta si lze všimnou rozdělení na řadu malých tříd a sedmi balíčků. Toto rozdělení bylo zvoleno na základě zkušeností s vývojem podobných grafických aplikací a snaží se pomáhat dobré orientaci v projektu. Náhled na architekturu je na Obrázku 25.

*Poznámka: V našem případě tvoří vrstvu Model balíček Service, obsahující třídy s aplikační logikou.*

[32] Dokumenace Spring: https://spring.io/docs
[33] Dokumentace Maven: https://maven.apache.org/index.html




Obrázek 25: Rozdělení na balíčky - HPC klient

- **View**: balíček obsahující soubory FXML[34] definující grafickou podobu aplikace, kaskádové styly a použité písma v aplikaci.
- **Controller**: balíček tvořený třídami zpracovávající uživatelské požadavky z grafického prostředí a následně volající příslušné metody na objektech z balíčku Service.
- **Service**: balíček tvořený třídami s hlavních logikou aplikace, rozdělený na dvě části remote a local. Remote pro vygenerované třídy z WSDL a local pro vlastní zpracování generovaného WSDL.
- **Task**: balíček složený ze sady tříd spouštěných ve vláknech umožňující aplikaci reagovat na uživatelské vstupy během zpracování požadavku.
- **Model**: balíček obsahující doménové objekty klienta jako Job, Session a konstanty použité v aplikaci.
- **DataFX**: balíček obsahujicící třídu nastavující grafickou knihovnu JavaFX.
- **Util**: balíček pomocných nástrojů nad rámem programovacího jazyka Java, jako například metoda isEmpty() ověřující jestli je textový řetězec prázdný, i když je proměnná null.

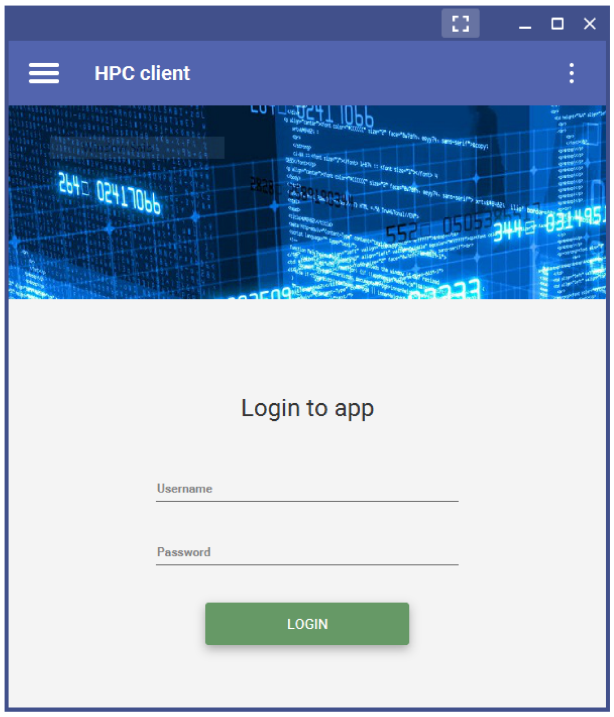
[34]Formát FXML je speciální forma XML formátu používající značky z HTML stránek rozšířená o JavaFX komponenty.

## 6.5 Pohled na aplikaci

### Přihlášení

Při zapnutí aplikace se vždy zobrazí výzva k přihlášení, pokud se uživatel přihlašuje znova aplikace si uloží naposledy použité uživatelské jméno a uživatel pak zadá jenom svoje heslo. Ukázka přihlašovací obrazovky je na Obrázku 26. K dalším vylepšením patří registrovaný posluchač na klávesu *Enter*, kdy uživatel po zadání hesla může rychle stisknout *Enter* a nemusí najíždět myší na tlačítko pro přihlášení.

Po technické stránce je zde použit návrhový vzor Client Session State[13] pro uložení tokenu získaného po úspěšném přihlášení. Token identifikuje uživatele a je nutné mít ho uchovaný po celou dobu běhu aplikace. Posluchač na klávesu *Enter* je realizován pomocí vzoru Observer[13]. Naposledy použité uživatelské jméno je uloženo do souboru *setting.xml*.




Obrázek 26: Přihlašovací obrazovka - HPC klient

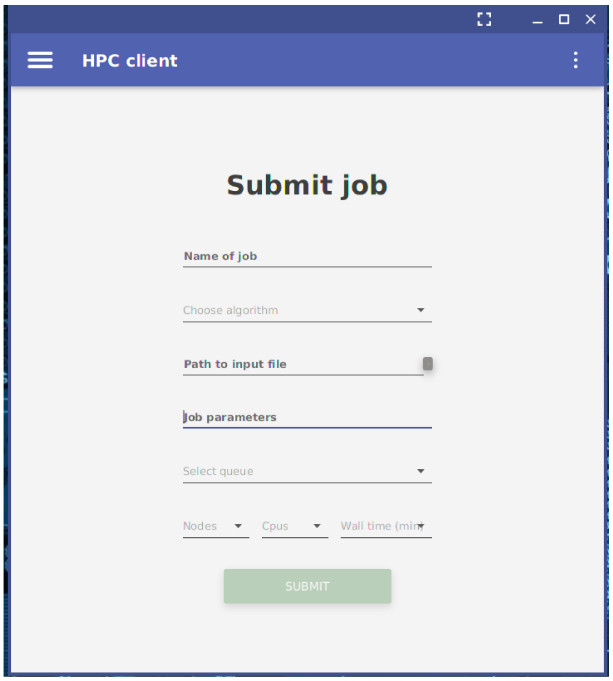
```
<userProfile>
        <userName>cag0008</userName>
        <positionWindowX>453</positionWindowX>
        <positionWindowY>48</positionWindowY>
        <lastLogin>11-03-2018</lastLogin>
</userProfile>
```

Výpis 34: Uživatelské nastavení pro HPC klienta

### Předložení úlohy

Aplikace je napojena oproti počítači Salomon kde si uživatel může vybrat ze tří algoritmů (WordCount, Invertovaného indexování a K-means) implementovaných ve Sparku. Před předložením musí uživatel vyplnit název úlohy, vybrat algoritmus, cestu k vstupnímu textovému souboru, frontu pro běh, počet výpočetních uzlů, počet procesorů a nakonec atribut WallTime[35] jak je vykresleno na Obrázku 27. Případně může uživatel ještě přidat parametry úlohy, pokud chce spustit algoritmus se speciálním nastavením. Před předložením chybných údajů chrání aplikaci real-time validace, aktivující tlačítko pro předložení úlohy, pokud jsou všechny vstupní pole zadané správně.

Z technického pohledu jsou validace realizovány pomocí druhého vlákna, které při každé změně vstupního políčka spustí validace a následně aktivaci nebo deaktivaci tlačítka pro předložení úlohy. Spouštění vláken v aplikaci bylo nutné monitorovat a řídit, proto byla vytvořena třída TaskPool přes kterou se spouští všechny ostatní vlákna kromě hlavního. Díky tomu, že máme v třídě TaskPool seznam všech aktuálně spouštěných vláken, můžeme jednoduše při přepnutí do jiné obrazovky provést ukončení vláken pro předchozí obrazovku.



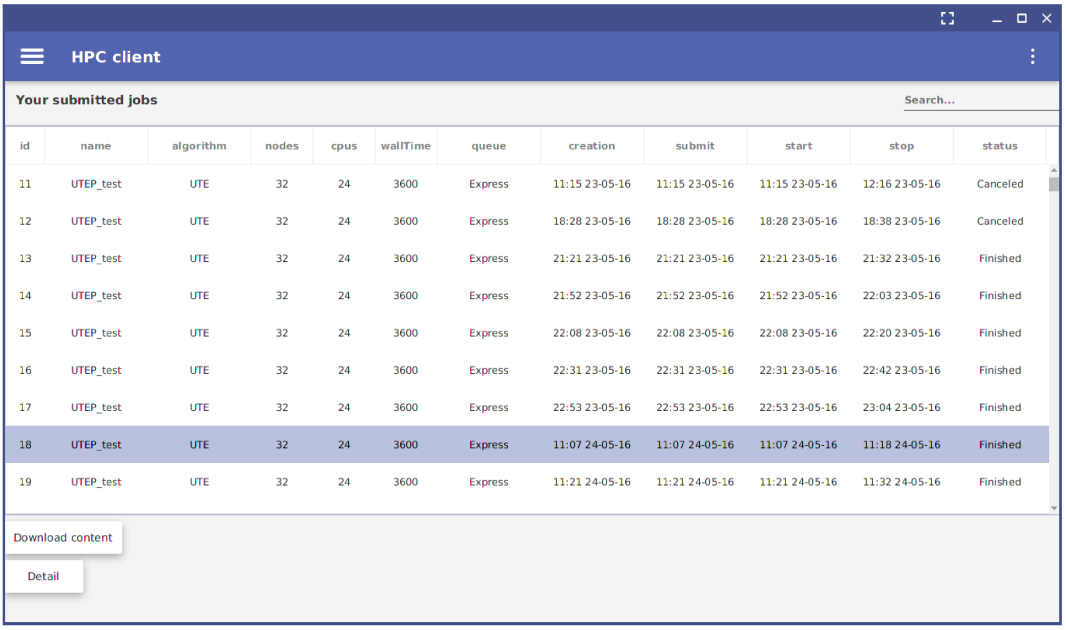

Obrázek 27: Obrazovka pro zadání úlohy - HPC klient

[35] WallTime je maximální čas jak dlouho může úloha běžet na superpočítači po přidělení zdrojů, pokud aplikace přesáhne tuto dobu je okamžitě ukončena.

### Zobrazení všech úloh

Aplikace zobrazuje v přehledné tabulce seznam naposledy předložených úloh s základními informacemi jak je zobrazeno na Obrázku 28. Pokud chce uživatel získat podrobnější informace, může po vybrání dané úlohy kliknout na tlačítko Detail zobrazující podrobné informace k úloze. Tabulka si sama automaticky aktualizuje stav úloh v pravidelných intervalech. Dále uživatel může přes vyhledávací políčko full-textově prohledávat celou tabulku a filtrovat záznamy. Pokud úloha už doběhla a je ve stavu FINISH, může uživatel stáhnout výsledky přes tlačítko Download content.

Pro automatickou aktualizaci stavů je v aplikaci vytvořené další vlákno, které každých 10 sekund volá službu HPC service pro získání všech předložených úloh daného uživatele. Poté se data synchronizují se speciální kolekcí nazvanou ObservableList[36], která je provázaná s tabulkou a při změně provede automaticky aktualizaci tabulky. Vyhledávání se provádí za využití filtrace kolekce záznamů v tabulce pomocí predikátů, procházející všechny záznamy v tabulce a pokud splňuje danou podmínku zůstane zachován, jinak je z kolekce vyřazen.




Obrázek 28: Obrazovka zobrazující všechny předložené úlohy - HPC klient

[36] Popis ObservableList: https://docs.oracle.com/javase/8/javafx/api/javafx/collections/ObservableList.html

### Ukázka zdrojového kódu

Vybrané ukázky zdrojového kódu z aplikace HPC klienta jsou zobrazeny ve výpisu 35 a 36.

```
public class TaskPool {

   private static Stack<CancelableTask> runningTasks = new Stack<>();

   public static void start(CancelableTask task){
       new Thread(task).start();
       runningTasks.add(task);
   }

   public static void cancelAllRunningTasks(){
       while (!runningTasks.empty()){
           CancelableTask task =runningTasks.pop();
           task.terminate();
       }
   }
}
```

Výpis 35: Třída TaskPool

```
private ChangeListener<String> setupSearchField(final JFXTreeTableView<
    JobFormattedForTable> tableView) {
       return (o, oldVal, newVal) ->
               tableView.setPredicate(jobProperty -> {
                   final JobFormattedForTable job = jobProperty.getValue();
                   return job.name.get().contains(newVal)
                           || job.algorithm.get().contains(newVal)
                           || job.status.get().contains(newVal)
                           || job.queuen.get().contains(newVal)
                           || job.id.get().contains(newVal);
               });
   }
```

Výpis 36: Implementace full-textového vyhledávání nad tabulkou

# 7 Měření

V rámci porovnání různých implementací algoritmů pro Hadoop a Spark na infrastruktuře HPC v technologickém centru IT4Innovations byly použity dvě datové sady. Pro textovou analýzu byla zvolena část anglické encyklopedie Wikipedia[37] a pro shlukovou analýzu byla vygenerována množina bodů obsahující 100 shluků.

## 7.1 Popis datových sad

Datová sada obsahující historickou verzi Wikipedie byla vybrána z důvodu velkého množství volně dostupného textu v řádech několika GB[38]. Dataset je složen z 30 částí, v našem experimentu bylo použito prvních 12 částí představujících 10 GB dat ve formátu XML. Před samotným měřením bylo nutné data připravit. Byla provedena filtrace jenom části atributů u jednotlivých záznamů, jednalo se o atributy název článku a textový obsah článku. Tím se vytvořil dataset čítající 7,5 GB. Nejdříve se atributy snažily extrahovat za použití výchozího převodníku XML dokumentů obsaženého v jazyce Python. Zde se ale narazilo na jeden velký problém, výchozí převodník se snaží nahrát celý XML dokument do operační paměti a poté ho začíná zpracovávat. Tento přístup není možné použít pro soubory čítající několik GB, proto byl nakonec implementován vlastní SAX[39] převodník načítající jednotlivé záznamy do operační paměti postupně. Při extrakci atributů nejsme nijak závislý na předchozích záznamech v dokumentu, a proto je SAX převodník ideálním řešením zmíněného problému.

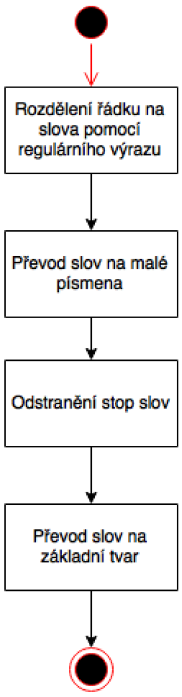
Pro shlukovou analýzu byla vytvořena množina bodů ve 2D prostoru pomocí vlastního generátoru, kde na vstupu algoritmu zadáme počet očekávaných shluků a celkový počet bodů v množině. Algoritmus nejdříve zapíše do souboru centra shluků definované na začátku a pak provádí náhodné přidělení zbytku bodů v okolí daného centra. Platí zde podmínka, že body jsou rozděleny rovnoměrně mezi všechny shluky. Generátorem lze snadno vytvářet velké množiny v řádu několika GB a rovněž dopředu víme výsledek, kolik shluků bude v množině a kde se nachází centra shluků. Tím si můžeme ověřit správnost implementace testovaných algoritmů. Pro experiment byla vytvořena množina se 100 shluky a 290 000 body čítající 9.9 GB.

## 7.2 Testované algoritmy

K testování byly implementovány tři algoritmy, dva na textovou analýzu a jeden pro shlukovou analýzu. První zástupce u textové analýzy je algoritmu WordCount, volně přeložený do češtiny jako čítač slov. Jedná se o velmi jednoduchý algoritmus, ale i tak poskytuje velmi zajímavé porovnání paralelizace technologií Hadoop a Spark. Druhým implementovaným algoritmem je Invertované indexování. Oba algoritmy přistupují k dokumentům stejně, čtou řádku po řádce a

[37] Internetová encyklopedie dostupná na https://cs.wikipedia.org/
[38] Archiv Wikipedie můžete nalézt na adrese: https://dumps.wikimedia.org/backup-index.html
[39] SAX parser: https://www.tutorialspoint.com/java_xml/java_sax_parser.htm

provádí převod řádky na slova. Převod se skládá z několika kroků, nejdříve se pomocí regulárního výrazu vyberou slova z řádky, poté se nalezené slova převedou na malé písmena. Následuje vyřazení anglických stop slov jako například: he, she, is, are, apod. Posledním krokem je převod slova na základní tvar za využití algoritmu Porter Stemming[40]. U shlukové analýzy byl použit algoritmus K-means. Podrobnější popis algoritmů lze najít v části 5. Implementované algoritmy. Všechny algoritmy byly zároveň implementované pro technologii Hadoop a Spark.




Obrázek 29: Předzpracování řádky u algoritmů WordCount a Invertovaného indexování

## 7.3 Testovací prostředi

Implementované algoritmy byly otestovány na superpočítači Anselm a to jak verze pro Hadoop, tak verze pro Spark. V prvotní fázi byl záměr otestovat algoritmy pro Hadoop na superpočítači Salomon, ale superpočítač Salomon nemá lokální diskové úložiště u každého výpočetního uzlu. Bohužel Salomon má jenom sdílený společný síťový disk a pokus nasadit Hadoop na tento typ úložiště způsoboval problémy. Po delším zkoumání chybových logů na serveru a článku na internetu se dospělo k závěru, že Hadoop na Salomon nasadit nepůjde, a proto byl k otestování zvolen superpočítač Anselm, kde byla doinstalována verze Hadoopu 1.2.1 a Sparku 2.1.2.

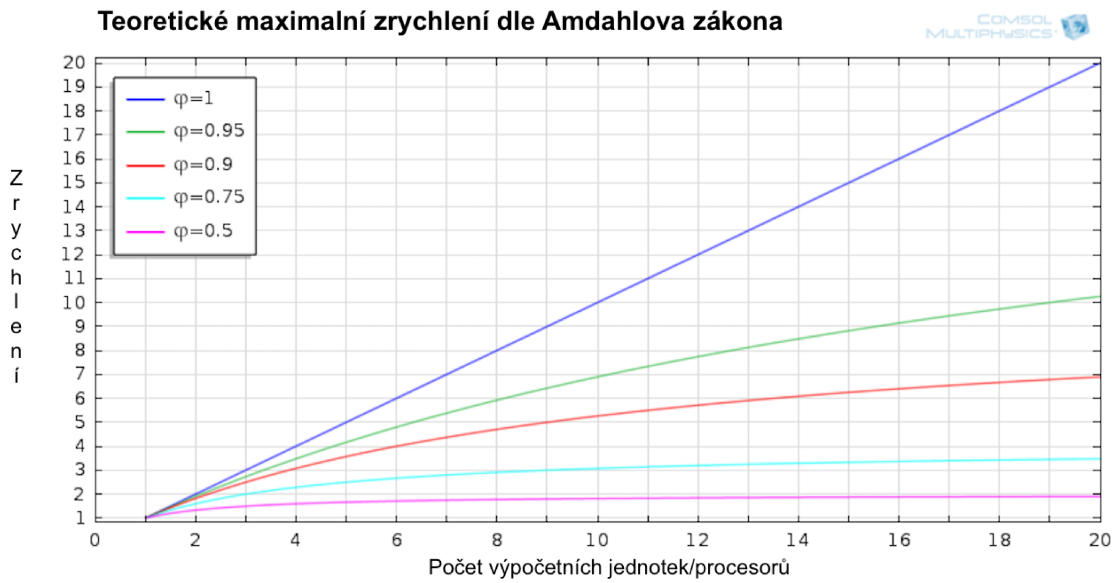
## 7.4 Průběh měření

Všechny měření byly provedeny nezávisle na sobě třikrát, aby se předešlo možné chybě v měření způsobenou například aktuálním přetížením výpočetního uzlu nebo sítě. U technologie Spark

[40] PorterStemmer algoritmus: https://tartarus.org/martin/PorterStemmer/

bylo nad každým algoritmem provedeno měření nad procesory a následně nad výpočetními uzly. Spark umožňuje i lokální paralelizaci algoritmů, díky možnosti provádět výpočet v operační paměti. U technologie Hadoop toto měření ale nemá význam, jelikož používá ke své činnosti distribuované diskové úložiště.

U paralelizace nad výpočetními uzly bylo testováno chování nad 2, 4 a 8 uzly. U procesorové paralelizace bylo postupně zkoumáno jak se chová algorimus nad 2,4,8 a 16 procesory. Výsledné časové hodnoty jsou zapsány v tabulkách, kde pro každou tabulku je vytvořen Amdahlův graf[28], pojednávající o tom jak moc ovlivní počet procesorů/výpočetních uzlů celkové zrychlení algoritmu.




Obrázek 30: Amdahlův graf[28]

$$S(P) = \frac{T(1)}{T(P)} = \frac{1}{\frac{\varphi}{P} + (1 - \varphi)} \tag{1}$$

$$S_{max}(\varphi) = \lim_{P \to \infty} S(P) = \frac{1}{1 - \varphi} \tag{2}$$

Ve vzorečku proměnná S představuje teoretické zrychlení, proměnná T čas v milisekundách, proměnná P vyjadřuje počet výpočetních procesorů nebo jednotek a $\varphi$ je konstanta zrychlení.

*Z grafu vyplývá, že při 95 procentní paralelizaci můžeme dosáhnout maximálně deseti násobného zrychlení a pak už nezáleží kolik výpočetních uzlů nebo procesorů máme zapojeno. Pokud je paralelizace pouze 75 procentní lze dosáhnout maximálně čtyřnásobného zrychlení[28].*

## 7.5 Výsledky

### 7.5.1 Algoritmus WordCount

První dvě tabulky porovnávají jak rychle zpracuje algoritmus WordCount vstupní dataset anglické Wikipedie ve formátu CSV čítající 7,5 GB nad skupinou výpočetních uzlů pomocí Hadoopu a Sparku. Každý výpočetní uzel si pro výpočet alokoval 16 procesorů Intel Xeon (E5-2670 2.6 Ghz). Tabulka číslo 3 zachycuje chování algoritmu při paralelizaci výpočtu nad procesory u technologie Spark, kde je použitá menší datová sada o velikosti 462 MB a výpočet je prováděn na jednom výpočetním uzlu.

| Počet jednotek | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 1513078 | 1520385 | 1499329 | 1510931 | 25,18 | 1,00 |
| 2 | 776976 | 823696 | 767555 | 789409 | 13,16 | 1,91 |
| 4 | 398982 | 408684 | 401126 | 402931 | 6,72 | 3,75 |
| 8 | 226370 | 225185 | 221730 | 224428 | 3,74 | 6,73 |

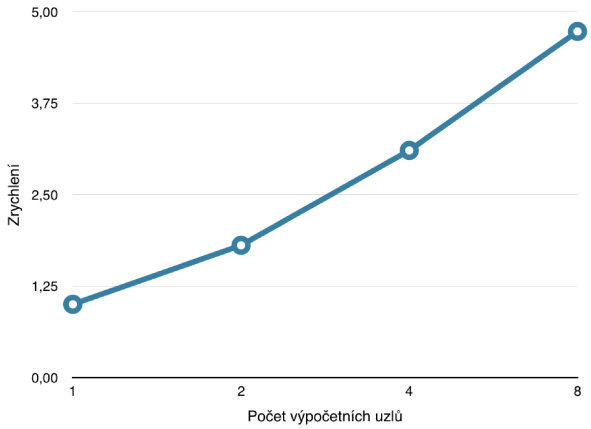
Tabulka 1: Měření algoritmu WordCount nad výpočetními uzly v Hadoopu (dataset 7,5 GB)

| Počet jednotek | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 1096991 | 1081620 | 1097231 | 1091947 | 18,20 | 1,00 |
| 2 | 681317 | 614397 | 600047 | 631920 | 10,53 | 1,73 |
| 4 | 378989 | 333535 | 329489 | 347338 | 5,79 | 3,14 |
| 8 | 197078 | 193566 | 194852 | 195165 | 3,25 | 5,59 |

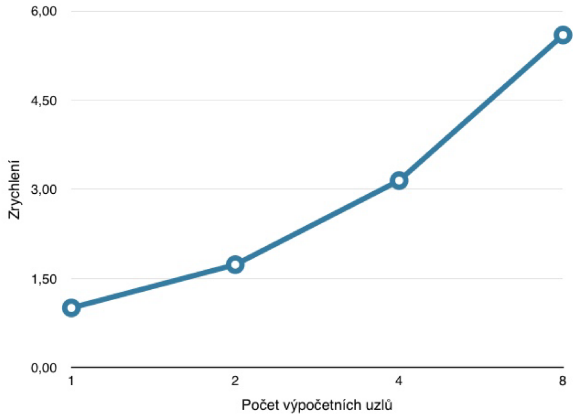
Tabulka 2: Měření algoritmu WordCount nad výpočetními uzly ve Sparku (dataset 7,5 GB)

| Počet CPU | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 2299949 | 2341261 | 2317416 | 2319542 | 38,66 | 1,00 |
| 2 | 1203189 | 1202966 | 1233873 | 1213343 | 20,22 | 1,91 |
| 4 | 659634 | 659283 | 663419 | 660779 | 11,01 | 3,51 |
| 8 | 334505 | 336090 | 333257 | 334617 | 5,58 | 6,93 |
| 16 | 186246 | 185978 | 182831 | 185018 | 3,08 | 12,54 |

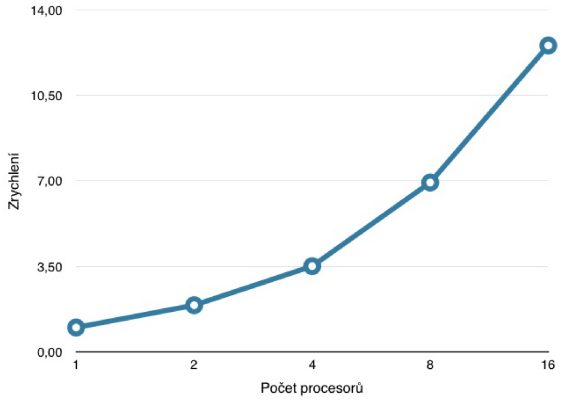
Tabulka 3: Měření algoritmu WordCount nad procesory ve Sparku (dataset 462 MB)




Zrychlení algoritmu WordCount implementovaného pro Hadoop na Anselmu (výpočetní uzly)




Zrychlení algoritmu WordCount implementovaného pro Spark na Anselmu (výpočetní uzly)




Zrychlení algoritmu WordCount implementovaného pro Spark na Anselmu (procesory)

### 7.5.2 Algoritmus Invertovaného indexování

První dvě tabulky porovnávají jak rychle zpracuje algoritmus Invertovaného indexování vstupní dataset anglické Wikipedie ve formátu CSV čítající 7,6 GB nad skupinou výpočetních uzlů pomocí Hadoopu a Sparku. Každý výpočetní uzel si pro výpočet alokoval 16 procesorů Intel Xeon (E5-2670 2.6 Ghz). Tabulka číslo 3 zachycuje chování algoritmu při paralelizaci výpočtu nad procesory u technologie Spark, kde je použitá menší datová sada o velikosti 462 MB a výpočet je prováděn na jednom výpočetním uzlu.

| Počet jednotek | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 2305203 | 2312947 | 2304079 | 2307410 | 38,46 | 1,00 |
| 2 | 1321525 | 1251202 | 1258668 | 1277132 | 21,29 | 1,81 |
| 4 | 751675 | 733360 | 745201 | 743412 | 12,39 | 3,10 |
| 8 | 483467 | 483207 | 496626 | 487767 | 8,13 | 4,73 |

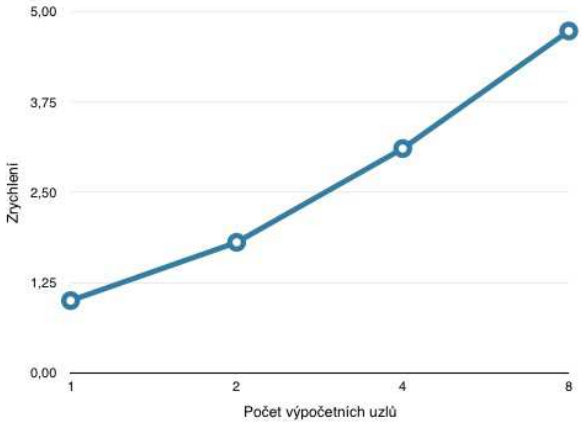
Tabulka 4: Měření algoritmu Invertovaného indexování nad výpočetními uzly v Hadoopu (dataset 7,6 GB)

| Počet jednotek | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 1208862 | 1216671 | 1226080 | 1217204 | 20,29 | 1,00 |
| 2 | 608035 | 738962 | 608257 | 651751 | 10,86 | 1,87 |
| 4 | 370108 | 372424 | 376038 | 372857 | 6,21 | 3,26 |
| 8 | 220378 | 225885 | 218327 | 221530 | 3,69 | 5,49 |

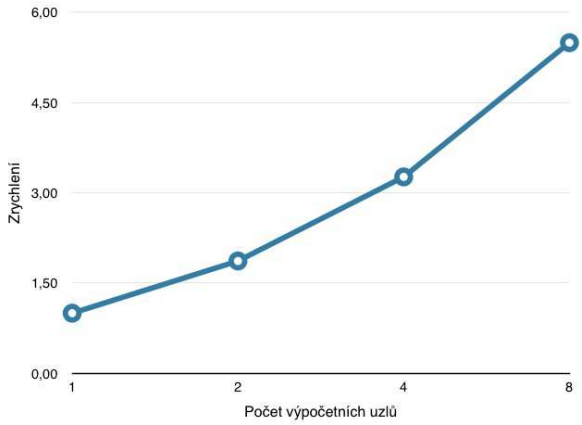
Tabulka 5: Měření algoritmu Invertovaného indexování nad výpočetními uzly ve Sparku (dataset 7,6 GB)

| Počet CPU | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 3140980 | 3073814 | 3076173 | 3096989 | 51,62 | 1,00 |
| 2 | 1573195 | 1549042 | 1597625 | 1573287 | 26,22 | 1,97 |
| 4 | 859994 | 853088 | 878850 | 863977 | 14,40 | 3,58 |
| 8 | 454956 | 462497 | 450838 | 456097 | 7,60 | 6,79 |
| 16 | 243452 | 246781 | 243499 | 244577 | 4,08 | 12,66 |

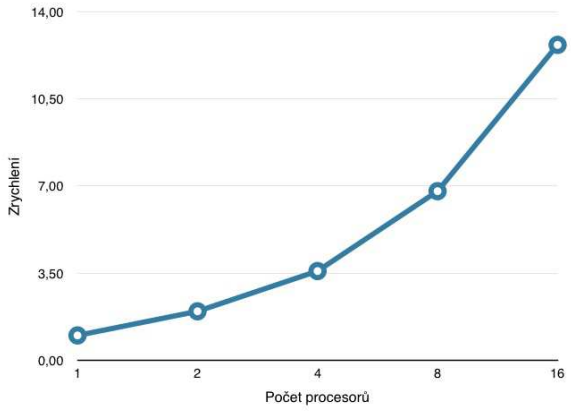
Tabulka 6: Měření algoritmu Invertovaného indexování nad procesory ve Sparku (dataset 462 MB)




Zrychlení Invertovaného indexování implementovaného pro Hadoop na Anselmu (výpočetní uzly)




Zrychlení Invertovaného indexování implementovaného pro Spark na Anselmu (výpočetní uzly)




Zrychlení Invertovaného indexování implementovaného pro Spark na Anselmu (procesory)

### 7.5.3 Algoritmus K-means

Pro měření K-means byla vygenerována množina 290 000 000 bodů s 100 shluky a velikostí 9,9 GB. Ukončovací podmínka K-means byla nastavena u všech implementací na maximálně 10 iterací algoritmu. Tabulka číslo 1 zachycuje měření vlastní implementace K-means pro technologii Hadoop testovanou pro různý počet výpočetních uzlů nad počítačem Anselm, přičemž každý výpočetní uzel si alokoval 16 procesorů Intel Xeon (E5-2670 2.6 Ghz). Tabulka číslo 2 a číslo 3 zobrazuje měření vlastní a Apache implementace K-means pomocí technologie Spark se stejnou datovou sadou nad výpočetními uzly Anselmu. Dále tabulky 4 a 5 zachycují chování algoritmu při paralelizaci výpočtu nad procesory u technologie Spark, kde je použita menší množina bodů s 17 000 000 body a 100 shluky o velikosti 451 MB a výpočet je prováděn na jednom výpočetním uzlu.

| Počet procesorů | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 3492218 | 3369672 | 3477496 | 3446462 | 57,44 | 1,00 |
| 2 | 2017404 | 1941606 | 1987934 | 1982315 | 33,04 | 1,74 |
| 4 | 1389274 | 1402893 | 1386441 | 1392869 | 23,21 | 2,47 |
| 8 | 824372 | 810536 | 813632 | 816180 | 13,60 | 4,22 |

Tabulka 7: Měření vlastní implementace K-means nad výpočetními uzly v Hadoopu (dataset 9,9 GB)

| Počet procesorů | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 3017897 | 2997278 | 3049360 | 3021512 | 50,36 | 1,00 |
| 2 | 1522381 | 1562844 | 1519421 | 1534882 | 25,58 | 1,97 |
| 4 | 907784 | 901439 | 922259 | 910494 | 15,17 | 3,32 |
| 8 | 592089 | 602037 | 577338 | 590488 | 9,84 | 5,12 |

Tabulka 8: Měření vlastní implementace K-means nad výpočetními ve Sparku (dataset 9,9 GB)

| Počet procesorů | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 867036 | 837047 | 857507 | 853863 | 14,23 | 1,00 |
| 2 | 457554 | 465009 | 453574 | 458712 | 7,65 | 1,86 |
| 4 | 252721 | 254547 | 254344 | 253871 | 4,23 | 3,36 |
| 8 | 166961 | 168732 | 168849 | 168181 | 2,80 | 5,08 |

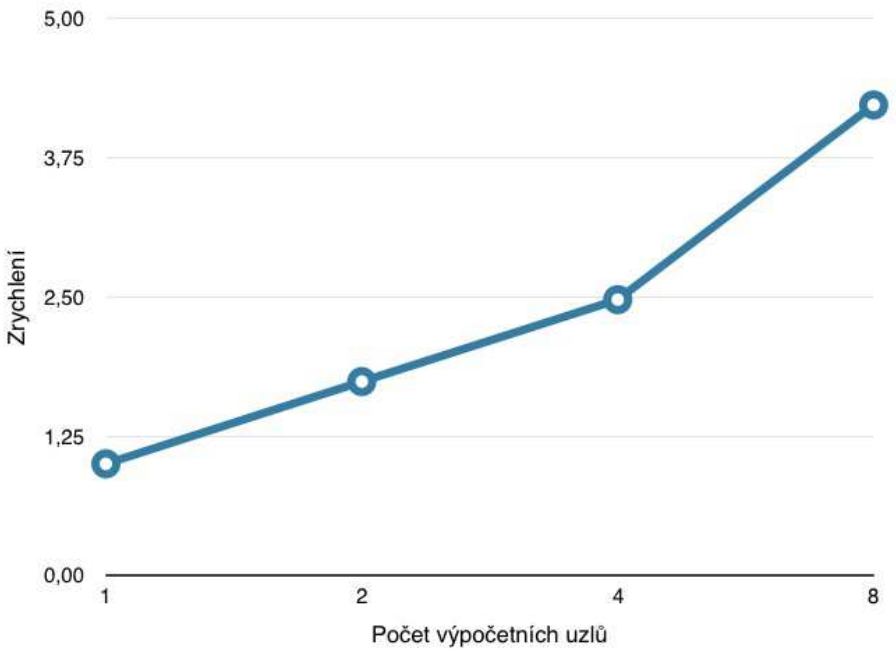
Tabulka 9: Měření implementace K-means od Apache nad výpočetními ve Sparku (dataset 9,9 GB)

| Počet CPU | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 2184740 | 2132179 | 2130322 | 2149080 | 35,82 | 1,00 |
| 2 | 1060663 | 1052381 | 1057410 | 1056818 | 17,61 | 2,03 |
| 4 | 568219 | 554299 | 572900 | 565139 | 9,42 | 3,80 |
| 8 | 283925 | 284739 | 293816 | 287493 | 4,79 | 7,48 |
| 16 | 179029 | 181200 | 186547 | 182259 | 3,04 | 11,79 |

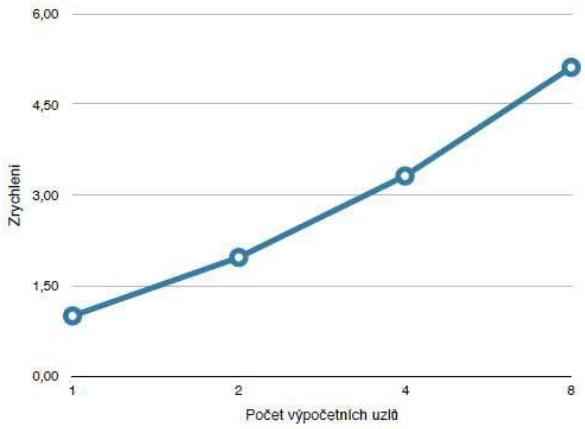
Tabulka 10: Měření vlastní implementace K-means nad procesory ve Sparku (dataset 451 MB)

| Počet CPU | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 291716 | 300261 | 295387 | 295788 | 4,93 | 1,00 |
| 2 | 148625 | 153397 | 152136 | 151386 | 2,52 | 1,95 |
| 4 | 83183 | 82571 | 82255 | 82670 | 1,38 | 3,58 |
| 8 | 56265 | 55293 | 57156 | 56238 | 0,94 | 5,26 |
| 16 | 42564 | 41644 | 43012 | 42407 | 0,71 | 6,98 |

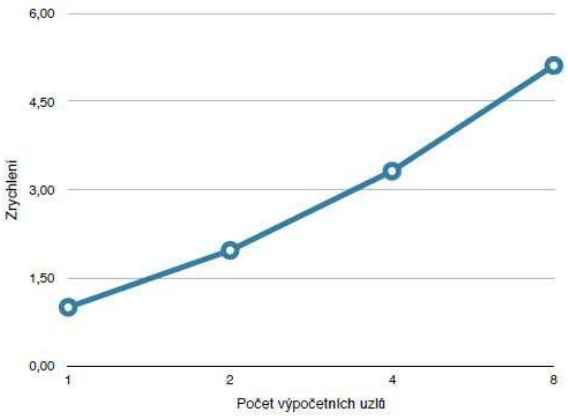
Tabulka 11: Měření implementace K-means od Apache nad procesory ve Sparku (dataset 451 MB)



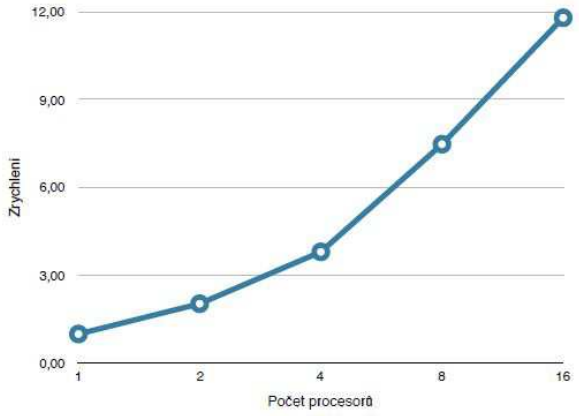

Zrychlení K-means implementovaného pro Hadoop na Anselmu (výpočetní uzly)



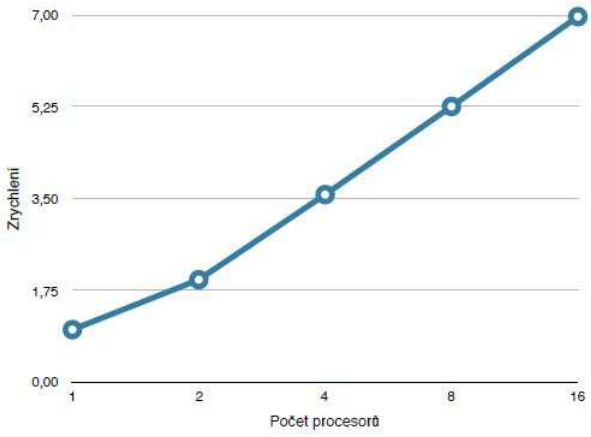

Zrychlení K-means implementovaného pro Spark na Anselmu (výpočetní uzly)




Zrychlení K-means od Apache pro Spark na Anselmu (výpočetní uzly)




Zrychlení K-means implementovaného pro Spark na Anselmu (procesory)




Zrychlení K-means od Apache na Anselmu (procesory)

### 7.5.4 Testování Sparku na osobním počítači

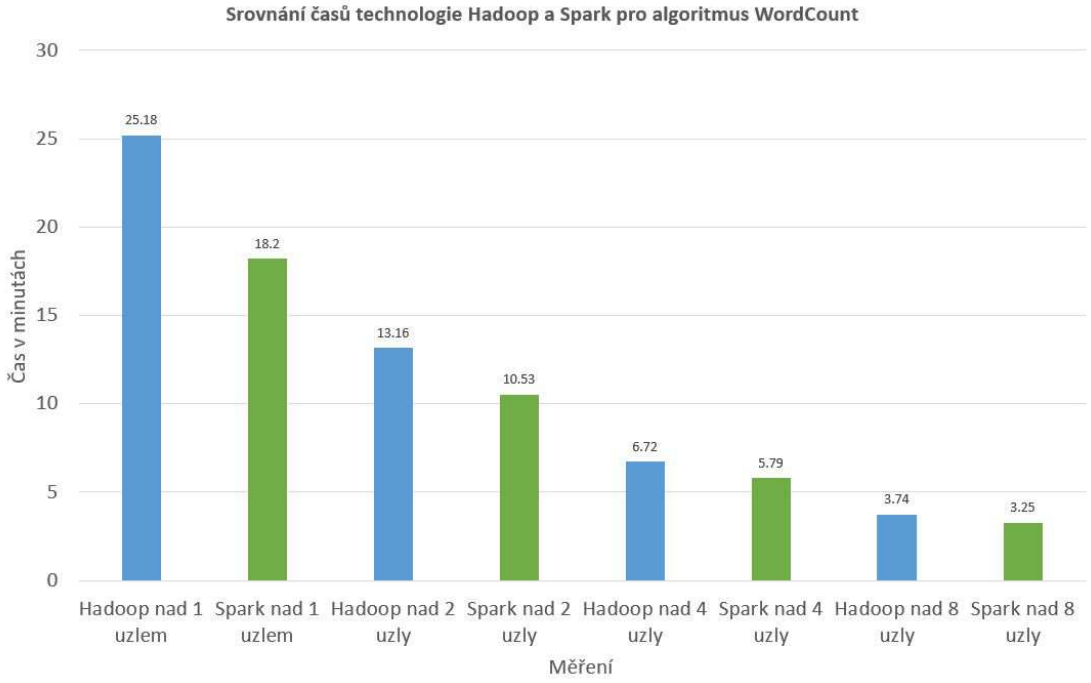
Nakonec bylo provedeno měření k jak velkému zrychlení výpočtu algoritmu WordCount dochází, při nasazení Sparku na osobní počítač s několika jádrovým procesorem. Jako osobní počítač byl zvolen Macbook Pro řady MGX72LL/A[41], který je osazený čtyřjádrových procesorem Intel core i5, 8 GB operační paměti a SSD diskem. Datovou sadu představovala část anglické Wikipedie ve formátu CSV čítající 462 MB, stejná jako v předchozím měření pro procesorovou paralelizaci nad Anselmem. Výsledky měření jsou zaznamenány v Tabulce číslo 12.

| Počet vláken | Čas měření 1 (ms) | Čas měření 2 (ms) | Čas měření 3 (ms) | Průměrný čas (ms) | Čas (minuty) | Zrychlení |
|---|---|---|---|---|---|---|
| 1 | 2941979 | 2878878 | 2906228 | 2909028 | 48,48 | 1,00 |
| 2 | 1492871 | 1474805 | 1525928 | 1497868 | 24,96 | 1,94 |
| 4 | 1376055 | 1395671 | 1386241 | 1385989 | 23,10 | 2,10 |

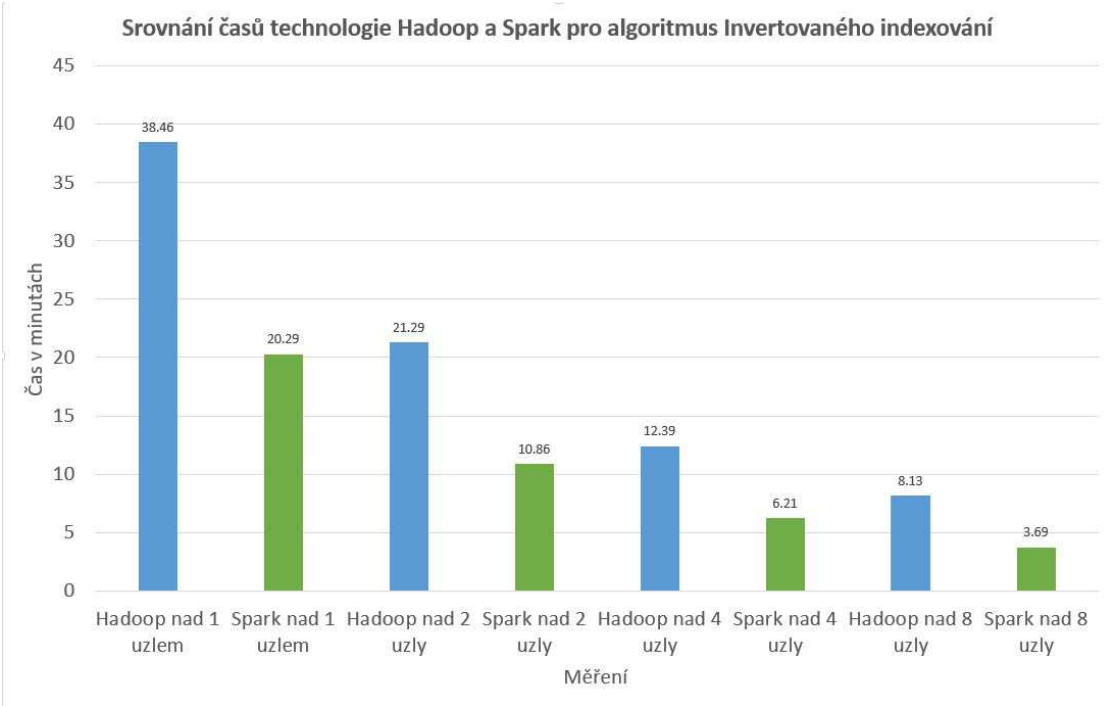
Tabulka 12: Měření algoritmu WordCount nad Macbook Pro ve Sparku (dataset 462 MB)

[41] Technická specifikace Macbook Pro: https://everymac.com/systems/apple/macbook_pro/specs/macbook-pro-core-i5-2.6-13-mid-2014-retina-display-specs.html

### 7.5.5 Srovnání technologie Hadoop vs Spark




Hadoop vs Spark (algoritmus WordCount)




Hadoop vs Spark (algoritmus Invertovaného indexování)

## 7.6 Rozbor výsledků

Měření potvrdilo, že dochází k zrychlení algoritmů při přidávání výpočetních uzlů u obou technologií. Například u algoritmu Invertované indexování dosahuje implementace pro Hadoop okolo 90 procentního zrychlení nad výpočetními uzly dle Amdahlova grafu. Stejný algoritmus ve Sparku dosahuje ještě většího zrychlení blížící se 95 procent nad výpočetními uzly. Největšího zrychlení dosahuje implementace algoritmu nad procesory ve Sparku, kde zrychlení dosahuje více jak 95 procent. To je dáno tím, že komunikace mezi procesory na jednom výpočetním uzlu je daleko rychlejší než synchronizace výpočetních uzlů po síti.

Z tabulek je patrné, že algoritmy implementované pro technologii Spark jsou rychlejší ve všech případech i o několik desítek procent, než ty samé algoritmy pro technologii Hadoop. Tento jev lze vysvětlit rozdílným ukládáním mezivýsledků u obou technologií. Zatímco Hadoop si každý mezivýsledek části Map a Reduce ukládá na disk, Spark si mezivýsledky ukládá v operační paměti, která je mnohonásobně rychlejší. Při měření zpracování času zpracování algoritmu WordCount nad výpočetními uzly, je Spark o 27 procent rychlejší, při provádění výpočtu nad jedním výpočetním uzlem, než ten samý algoritmus v Hadoopu. Při zvyšování počtu výpočetních jednotek se časový rozdíl zpracování snižuje, kdy nad 8 výpočetními uzly je Spark rychlejší už jenom o 13,10 procent. U algoritmu Invertovaného indexování je rozdíl mezi implementací pro Hadoop a Spark ještě znatelnější. Při meření času zpracování algoritmu nad výpočetními uzly, dosahuje Spark zrychlení 47,24 procent oproti implementaci algoritmu pro Hadoop nad jedním výpočetním uzlem. A ani při osmi výpočetních uzlech nedochází k velkému snižování rozdílu, zde stále běží Spark o 45 procent rychleji. Nejvíce variant měření bylo u algoritmu K-means, kde vlastní implementace pro Spark i Hadoop nad jedním výpočetním uzlem běží skoro stejně dlouho, a až nad více výpočetními uzly Spark zase předbíhá Hadoop v rychlosti zpracování a to o 27,65 procent nad 8 výpočetními uzly. U technologie Spark byla porovnána i rychlost zpracování vlastní implementace algoritmu s implementací K-means od Apache. Vlastní implementace je v porovnání s implementací od Apache, cca 3,5 krát pomalejší. V upravené verzi Hadoopu myHadoop pro prostředí superpočítače se nepodařilo zprovoznit knihovnu Mahout obsahující implementaci K-means, proto byla otestována jenom vlastní implementace nad Hadoopem.

Nakonec bylo provedeno měření jak si vede Spark nad osobním počítačem obsahující čtyřjádrový procesor. Z měření vyplývá, že dosahuje cca dvojnásobného zrychlení nad dvěmi jádry oproti běhu nad jedním jádrem. Když byl algoritmus spuštěn nad čtyřmi jádry, bylo dosaženo pouze malého zrychlení oproti měření času nad dvěma jádry. To je způsobeno tím, že procesor obsažený v testovaném počítači je sice čtyřjádrový, ale dvě vlákna má pouze virtuální. Myslím si, že obě technologie jsou dobré pro distribuované zpracování velkých datových sad. Spark ale má v dnešní době jasně navrch, ať už jednoduším zápisem algoritmů v Pythonu, rozšířením programového modelu MapReduce o další vysokoúrovňové metody a hlavně rychlostí.

# 8 Závěr

V této diplomové práci byla popsaná technologie Apache Hadoop a Spark využíváné k distribuovaným výpočtům, následně byly technologie nasazeny na prostředí superpočítačů v technologickém centru IT4Innovations. Technologii Spark se sice podařilo nasadit na výkonnější superpočítač Salomon, ale u technologie Hadoop se při nasazování na Salomon narazilo na problém s nedostupností lokálního úložiště pro výpočetní uzly, proto musel být použit superpočítač Anselm kde se podařilo nasadit obě technologie a provést měření. Obě technologii jsou ale určeny pro nasazení jak na klasické počítače propojené do clusteru, tak pro svět HPC. Pro prostředí superpočítače se muselo u obou technologií provést složitější konfigurování, které bylo podrobně popsané v kapitolách této práce.

Hlavním cílem bylo porovnání různých implementací algoritmů s využitím Hadoopu a Sparku nad rozsáhlými datovými sadami v superpočítačovém centru. K otestování byly implementovány tři algoritmy pro obě technologie, jednalo se o WordCount, Invertované indexování pro textovou analýzu a K-means, představitel machine learning algoritmu, pro shlukovou analýzu. Následně byla provedena měření nad superpočítači Anselm. Z výsledků je patrné, že technologie Spark v rychlosti zpracování převyšuje technologii Hadoop, je to způsobeno hlavně tím, že u technologii Spark se ukládají mezivýsledky do operační paměti výpočetního uzlu, zatímco Hadoop každý mezivýsledek části Map nebo Reduce ukládá na pevný disk. U algoritmu K-means bylo porovnáni jak rychlá je moje vlastní implementace ve Sparku oproti Apache implementaci ve Sparku. Bylo zjištěno, že vnitřní implementace je cca 3.5 krát rychlejší při provádění výpočtu nad výpočetními uzly a cca 7 krát rychlejší při provádění výpočtu nad procesory, proto je rozumnější použít už hotové algoritmy pokud to je možné. V dnešní době bych doporučil použít technologii Spark místo Hadoopu, za prvé je rychlejší, ale navíc dává programátorovi na výběr mezi několika programovacími jazyky a rozšiřuje model MapReduce o další operace čímž zjednodušuje psaní kódu.

V závěru byl implementován grafický klient pro spouštění implementovaných algoritmů. Klient umožňuje spouštění úloh a monitorování předložených úloh na superpočítači Salomon. Komunikace se superpočítačem probíhá přes webovou službu SOAP za použití služby HPC as a Service. Cílem klienta byl i pěkný vzhled a snadné ovládání, proto byl zvolen dnes velmi oblíbený Material Design od Googlu. K čitelnosti kódu přispívá jeho rozdělení do balíčků a použití návrhových vzorů, čímž se aplikaci dává dobrý základ pro budoucí rozšíření a spouštění i dalších algoritmů nad superpočítačem.

# Literatura


[1] *Itbiz: Big Data: příležitost, nebo hrozba pro IT* [online]. 2012 [cit. 2017-12-18]. Dostupné z: http://www.itbiz.cz/clanky/big-data-spis-prilezitost-nezli-hrozba-pro-it

[2] Philip Russom. *Big Data Analytics*. TDWI RESEARCH, 2011.

[3] LAM, Chuck. *Hadoop in action*. United States of America: Manning Publications, 2011. ISBN 9781935182191.

[4] ZEČEVIĆ, Petar a Marko BONAĆI. *Spark in action*. United States of America: Manning Publications, 2016. ISBN 978-161-7292-606.

[5] *Introduction to Supercomputing - Jan Verschelde* [online]. 2016 [cit. 2017-11-18]. Dostupné z: http://homepages.math.uic.edu/ jan/mcs572/MCS572.pdf

[6] *Hadoop Tutorial: Introduction* [online]. 2012 [cit. 2017-08-20]. Dostupné z: https://www.tutorialspoint.com/hadoop/

[7] *Hadoop documentation: YARN design* [online]. [cit. 2017-12-03]. Dostupné z: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[8] *Hadoop documentation: HDFS design* [online]. [cit. 2017-12-03]. Dostupné z: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[9] MINER, Donald Miner a Adam SHOOK. *MapReduce Design Patterns*. O'Reilly Media, 2012. ISBN 1449327176.

[10] *Apache Spark: Introduction* [online]. 2017 [cit. 2017-08-20]. Dostupné z: https://www.tutorialspoint.com/apache_spark/

[11] HAN, Jiawei, Micheline KAMBER a Jian PEI. *Data mining: concepts and techniques*. 3rd ed. Haryana, India ; Burlington, MA: Elsevier, 2012. ISBN 93-809-3191-3.

[12] SHARAN, Kishori. *Learn JavaFX 8: building user experience and interfaces with Java 8*. United States of America: Apress, 2015. Expert's voice in Java. ISBN 14-842-1143-X.

[13] FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, 2003. ISBN 03-211-2742-0.

[14] KAMENIK, Pavel. *Příkazový řádek v Linuxu*. Olomouc: COMPUTER PRESS, 2012. ISBN 9788025139912.

[15] *IT4Innovations Documentation: General* [online]. 2017 [cit. 2017-08-20]. Dostupné z: https://docs.it4i.cz

[16] *PBS Professional: Documentation* [online]. 2017 [cit. 2017-12-18]. Dostupné z: https://pbsworks.com/pdfs/PBSUserGuide14.2.pdf

[17] Kaushik Velusamy, Deepthi Venkitaramanan, Nivetha Vijayaraju, Greeshma Suresh and Divya Madhu, "Inverted Indexing In Big Data Using Hadoop Multiple Node Cluster" International Journal of Advanced Computer Science and Applications(IJACSA), 4(11), 2013. http://dx.doi.org/10.14569/IJACSA.2013.041122

[18] DYER, Chris a Jimmy LIN. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010. ISBN 9781608453429.

[19] Zhao W., Ma H., He Q. (2009) Parallel K-Means Clustering Based on MapReduce. In: Jaatun M.G., Zhao G., Rong C. (eds) Cloud Computing. CloudCom 2009. Lecture Notes in Computer Science, vol 5931. Springer, Berlin, Heidelberg

[20] LUKASOVÁ, Alena a Jana ŠARMANOVÁ. *Metody shlukové analýzy*. Praha: Státní nakladatelství technické literatury, 1985. Metody shlukove analyzy

[21] *Electronic Journal for History of Probability and Statistics: K-means algorithm in cluster* [online]. [cit. 2017-12-03]. Dostupné z: http://www.jehps.net/Decembre2008/Bock.pdf

[22] *Introduction to K-means Clustering* [online]. 12.06.2016 [cit. 2018-03-25]. Dostupné z: https://www.datascience.com/blog/k-means-clustering

[23] *OBIEE 12c Advanced Analytic part 4: Cluster* [online]. 2016 [cit. 2018-03-25]. Dostupné z: http://obieeil.blogspot.cz/2016/01/obiee-12c-advanced-analytic-part-4.html

[24] KOSMÁK, Ladislav a Radovan POTŮČEK. *Metrické prostory*. Praha: Academia, 2004. ISBN 80-200-1202-8.

[25] *K-Means Clustering in Map Reduce* [online]. 2011 [cit. 2018-03-25]. Dostupné z: http://horicky.blogspot.cz/2011/04/k-means-clustering-in-map-reduce.html

[26] PEŠATOVÁ, Karina, Barbora POLÁKOVÁ a John CAWLEY, ed. *Supercomputing in science and engineering: IT4Innovations National Supercomputing Center, Czech Republic 2017*. Edition: 1st. Ostrava: VŠB - Technical University of Ostrava, 2017. ISBN 978-80-248-4037-6.

[27] SCHILDT, Herbert. *Java 7: výukový kurz*. Brno: Computer Press, 2012. ISBN 978-80-251-3748-2.

[28] *COMSOL BLOG: Added Value of Task Parallelism in Batch Sweeps* [online]. 2014 [cit. 2017-08-20]. Dostupné z: https://www.comsol.com/blogs/added-value-task-parallelism-batch-sweeps/

[29] *PBS documentation: Running a Job on HPC using PBS* [online]. 2018 [cit. 2018-01-01]. Dostupné z: https://hpcc.usc.edu/support/documentation/running-a-job-on-the-hpcc-cluster-using-pbs/

# A Zdrojový kód WordCount Hadoop

```
public class WordCountMapper extends Mapper<Object,Text,Text,IntWritable>{

    private final IntWritable ONE = new IntWritable(1);

    @Override
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String[] words = value.toString().trim().split(" ");
        for(String word: words){
            if (isWord(word)){
                String basicWord = stem(word.toLowerCase());
                if(!STOP_WORDLS.contains(basicWord)
                    context.write(new Text(basicWord),ONE);
            }
        }
    }

    private boolean isWord(String text){
        Pattern pattern = Pattern.compile("^[a-zA-Z]+$");
        Matcher matcher = pattern.matcher(text);
        return matcher.find();
    }

    private String stem(String word){
        Stemmer stemmer = new Stemmer();
        stemmer.add(word.toCharArray(),word.length());
        stemmer.stem();
        return new String(stemmer.toString());
    }

}

public class WordCountReducer extends Reducer<Text,IntWritable,Text,IntWritable>{

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        int sum = 0;
        for(IntWritable value: values){
            sum += value.get();
        }
        context.write(key,new IntWritable(sum));
    }
}
```

# B Zdrojový kód WordCount Spark

```
# Prepare
inputFile="input.txt";
outputDirectory="result";
hostname=socket.gethostname();
if os.path.exists(outputDirectory):
   shutil.rmtree(outputDirectory);

# Computation
startTime = int(round(time.time() * 1000));

def isWord(word):
   reg = re.compile('^[a-zA-Z]+$');
   return reg.match(word);

sc = SparkContext("local") # singe node on local
#sc = SparkContext("spark://"+hostname+":7077") # HPC clusters
lines = sc.textFile(inputFile);
countWords = lines.flatMap(lambda line: line.split(" ")) \
   .filter(lambda word: isWord(word)) \
   .map(lambda word: stem(word.lower())) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda x, y: x + y ) \
    .sortByKey(True);
countWords.saveAsTextFile(outputDirectory);

stopTime = int(round(time.time() * 1000));
print("Total time computation: " + str(stopTime-startTime) + " ms");
```

# C Zdrojový kód Invertované indexování Hadoop

```
public class InversionIndexMapper extends Mapper<Object, Text, Text, TotalOccurrence> {

    private final IntWritable numberOne = new IntWritable(1);

    /**
     * Function divide line from file into words and save to map in format:
     * <word,<totally word occurence, document number, docuemnt occurence>>
     */
    @Override
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        String[] items = line.split(",");

        int documentNumber = Integer.parseInt(items[0]);
        String documentContext = items[2];

        String[] words = documentContext.trim().split(" ");

        //map occurence word to shape:
        //<'some word',[total count,document count, document number]>
        for (String word : words) {
            if(isWord(word)){
                String basicWord = stem(word.toLowerCase());
                if(!STOP_WORDS.contains(basicWord)){
                    DocumentOccurrence documentOccurrence = new DocumentOccurrence(documentNumber, 1);
                    TotalOccurrence totalOccurrence = new TotalOccurrence();
                    totalOccurrence.setTotalCount(numberOne);
                    List<DocumentOccurrence> list = new ArrayList<>();
                    list.add(documentOccurrence);
                    totalOccurrence.setDocumentOccurences(list);
                    context.write(new Text(basicWord), totalOccurrence);
                }
            }
        }
    }

    private boolean isWord(String text){
        Pattern pattern = Pattern.compile("^[a-zA-Z]+$");
        Matcher matcher = pattern.matcher(text);
        return matcher.find();
    }

    private String stem(String word){
        Stemmer stemmer = new Stemmer();
        stemmer.add(word.toCharArray(),word.length());
        stemmer.stem();
        return new String(stemmer.toString());
    }
}
```

```java
public class InversionIndexReducer extends Reducer<Text, TotalOccurrence, Text, TotalOccurrence> {

    private Map<Integer, Integer> map;//<documentNumber, count occurrence>

    /**
     * Reduction class unite map value for word. Example: word "pepa" ->
     * <1,1,1>,<1,1,,1>,<1,2,1> -> <3,<1,2>,<2,1>>
     */
    @Override
    public void reduce(Text key, Iterable<TotalOccurrence> values, Context context) throws IOException, InterruptedException {
        map = new HashMap();
        int totalCount = 0;
        for (TotalOccurrence totalOccurrence : values) {
            for (DocumentOccurrence documentOccurrence : totalOccurrence.getDocumentOccurences()) {
                int documentNumber = documentOccurrence.getDocumentNumber().get();
                int count = documentOccurrence.getCount().get();
                saveToMap(documentNumber, count);
                totalCount += count;
            }
        }

        TotalOccurrence totalOccurrence = new TotalOccurrence();
        totalOccurrence.setTotalCount(totalCount);
        totalOccurrence.setDocumentOccurences(convertMap());
        context.write(key, totalOccurrence);
    }

    private void saveToMap(int documentNumber, int count) {
        if (map.containsKey(documentNumber)) {
            map.put(documentNumber, map.get(documentNumber) + count);
        } else {
            map.put(documentNumber, count);
        }
    }

    private List<DocumentOccurrence> convertMap() {
        List<DocumentOccurrence> result = new ArrayList<DocumentOccurrence>();
        for(Map.Entry<Integer,Integer> entry:map.entrySet()){
            DocumentOccurrence documentOccurrence = new DocumentOccurrence();
            documentOccurrence.setDocumentNumber(entry.getKey());
            documentOccurrence.setCount(entry.getValue());
            result.add(documentOccurrence);
        }
        return result;
    }
}
```

# D Zdrojový kód Invertované indexování Spark

```
# Computation
startTime = int(round(time.time() * 1000));

def inversionIndex(sc,inputFile,outputDirectory):
    lines = sc.textFile(inputFile)
    lines.flatMap(lambda line: lineToWords(line)) \
        .groupByKey() \
        .map(lambda item: (item[0], list(item[1]))) \
        .map(decideTotalCount) \
        .map(lambda item : swap1(item)) \
        .sortByKey(False) \
        .map(lambda item : swap2(item)) \
        .saveAsTextFile(outputDirectory)

def lineToWords(line):
    """
    Function read lines from CSV format (articleId, articleName, content) and transform
    to form [(word,(articleId,count))]

    for example:
        input: "1,cars,text text dog"
        output: [(text,(1,2)),(dog,(1,2))]
    """
    article = line.split(',')
    articleId = article[0]
    articleContent = article[2]

    wordcount={}
    for word in articleContent.split():
        if re.compile('^[a-zA-Z]+$').match(word):
            baseWord = stem(word.lower())
            if baseWord not in wordcount:
                wordcount[baseWord] = 1
            else:
                wordcount[baseWord] += 1

    result = []
    for baseWord,count in wordcount.items():
        result.append((baseWord,(articleId,count)))
    return result

def decideTotalCount(item):
    """
    Function calculate optional information, total count per all document occurrence.
    :param item: list <word,{(doc_num,doc_occurrence),..}>,..
    :return: list <word,{total_occurrence,[(doc_num,doc_occurrence),..]}
    """
    key = item[0]
    values = item[1]
    total_count = 0
    for value in values:
        total_count += value[1]
    return (key, (total_count, values))

def swap1(item):
    key = item[0]
    total_count = item[1][0]
    values = item[1][1]
    return (total_count,(key, values))

def swap2(item):
    total_count = item[0]
    key = item[1][0]
    values = item[1][1]
    return (key, (total_count,values))



inversionIndex(sc,inputFile,outputDirectory);
stopTime = int(round(time.time() * 1000));
print("Total time computation: " + str(stopTime-startTime) + " ms");
```

# E Zdrojový kód K-means Hadoop

```java
/**
* First iteration, k-random centers, in every follow-up iteration we have new calculated centers
*/
public class KMeansMapper extends Mapper<ClusterCenter, VectorWritable, ClusterCenter, VectorWritable> {

   private final List<ClusterCenter> centers = new ArrayList<>();
   private DistanceMeasurer distanceMeasurer;

   @SuppressWarnings("deprecation")
   @Override
   protected void setup(Context context) throws IOException, InterruptedException {
      super.setup(context);
      Configuration conf = context.getConfiguration();
      Path centroids = new Path(conf.get("centroid.path"));
      FileSystem fs = FileSystem.get(conf);

      try (SequenceFile.Reader reader = new SequenceFile.Reader(fs, centroids, conf)) {
         ClusterCenter key = new ClusterCenter();
         IntWritable value = new IntWritable();
         int index = 0;
         while (reader.next(key, value)) {
            ClusterCenter clusterCenter = new ClusterCenter(key);
            clusterCenter.setClusterIndex(index++);
            centers.add(clusterCenter);
         }
      }
      distanceMeasurer = new ManhattanDistance();
   }

   @Override
   protected void map(ClusterCenter key, VectorWritable value, Context context) throws IOException,
         InterruptedException {

      ClusterCenter nearest = null;
      double nearestDistance = Double.MAX_VALUE;
      for (ClusterCenter c : centers) {
         double dist = distanceMeasurer.measureDistance(c.getCenterVector(), value.getVector());
         if (nearest == null) {
            nearest = c;
            nearestDistance = dist;
         } else {
            if (nearestDistance > dist) {
               nearest = c;
               nearestDistance = dist;
            }
         }
      }
      context.write(nearest, value);
   }

}
```

```java
/**
* Calculate a new clustercenter for these vertices
*/
public class KMeansReducer extends Reducer<ClusterCenter, VectorWritable, ClusterCenter, VectorWritable> {

   public static enum Counter {
      CONVERGED
   }

   private final List<ClusterCenter> centers = new ArrayList<>();

   @Override
   protected void reduce(ClusterCenter key, Iterable<VectorWritable> values, Context context) throws IOException,
         InterruptedException {

      List<VectorWritable> vectorList = new ArrayList<>();
      DoubleVector newCenter = null;
      for (VectorWritable value : values) {
         vectorList.add(new VectorWritable(value));
         if (newCenter == null)
            newCenter = value.getVector().deepCopy();
         else
            newCenter = newCenter.add(value.getVector());
      }

      newCenter = newCenter.divide(vectorList.size());
      ClusterCenter center = new ClusterCenter(newCenter);
      centers.add(center);
      for (VectorWritable vector : vectorList) {
         context.write(center, vector);
      }

      if (center.converged(key))
         context.getCounter(Counter.CONVERGED).increment(1);

   }

   @SuppressWarnings("deprecation")
   @Override
   protected void cleanup(Context context) throws IOException, InterruptedException {
      super.cleanup(context);
      Configuration conf = context.getConfiguration();
      Path outPath = new Path(conf.get("centroid.path"));
      FileSystem fs = FileSystem.get(conf);
      fs.delete(outPath, true);
      try (SequenceFile.Writer out = SequenceFile.createWriter(fs, context.getConfiguration(), outPath,
            ClusterCenter.class, IntWritable.class)) {
         final IntWritable value = new IntWritable(0);
         for (ClusterCenter center : centers) {
            out.append(center, value);
         }
      }
   }
}
```

# F Zdrojový kód K-means Spark

```
# Prepare
inputFile = sys.argv[1]
outputDirectory = sys.argv[2]
environment = sys.argv[3] # LOCAL or SALOMAN
count_clusters = sys.argv[4]
maxIteration = sys.argv[5]

sc = ""; # SparkContext

if os.path.exists(outputDirectory):
    shutil.rmtree(outputDirectory)

if environment == "SALOMAN":
    hostname = socket.gethostname()
    sc = SparkContext("spark://" + hostname + ":7077") # HPC clusters
else:
    sc = SparkContext("local") # singe node on local

# Computation
startTime = int(round(time.time() * 1000))

def parseVector(line):
    return np.array([float(x) for x in line.split(' ')])


def closestPoint(p, centers):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(centers)):
        tempDist = np.sum((p - centers[i]) ** 2)
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex


lines = sc.textFile(inputFile)
data = lines.map(parseVector).cache()
K = int(count_clusters)
kPoints = data.takeSample(False, K, 1)
iteration = 0

while iteration < maxIteration:
    closest = data.map(
        lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(
        lambda st: (st[0], st[1][0] / st[1][1])).collect()
    for (iK, p) in newPoints:
        kPoints[iK] = p
    iteration=iteration+1

print("Final centroids: " + str(kPoints))

stopTime = int(round(time.time() * 1000))
print("Total time computation: " + str(stopTime - startTime) + " ms")
```

# G Obsah CD

- Elektronická verze diplomové práce ( PDF a latex formát)
- Zdrojový kód WordCount pro (Hadoop/Spark)
- Zdrojový kód Inverzního indexování pro (Hadoop/Spark)
- Zdrojový kód K-means pro (Hadoop/Spark)
- Grafický klient pro předložení úloh na superpočítač
- Pomocné programy napsané v Pythonu k stažení a přípravě datových sad